Clustering Dependencies over Relational Tables

by

Yuchen Gao

A thesis
presented to the University Of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
In
Computer Science

Waterloo, Ontario, Canada, 2015

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Integrity constraints have proven to be valuable in the database field. Not only can they help schema design (functional dependencies, FDs [1][2]), they can also be used in query optimization (ordering dependencies, ODs [4][5][8][9]), or data cleaning (conditional functional dependencies, CFDs [12] and denial constraints, DCs [14]). In this thesis, however, we will introduce a new type of integrity constraint, called a **clustering dependency (CD)**.

Similar to ordering dependencies which rely on the database operation **ORDER BY**, clustering dependencies focus on studying the operation **GROUP BY**. Furthermore, we claim that clustering dependencies are useful not only in query optimization as most integrity constraints do, but also useful in data visualization, data analysis and MapReduce.

In this thesis, we first introduce some examples of clustering dependencies in a real-life dataset. We then formally define clustering dependencies and elaborate on our motivation. We will also look into the reasoning system for clustering dependencies including the implication problem, consistency problem and influence rules for clustering dependencies. After that, we will propose two algorithms for clustering dependencies, first a checking algorithm that is able to check if a given dependency is valid in a table within $O(N * M)$ time, with N being the number of rows and M being the size of potentially aggregated attributes, a.k.a, the size of the right-hand-side attributes. Secondly, we propose a mining algorithm that is able to discover all potential clustering dependencies occurring in a table. Finally, we will use both synthetic and real-life data to test the performance of our mining algorithm.

Acknowledgements

First and foremost, I would like to give my most sincere gratitude to my supervisor, Prof. Grant Weddell. He granted me the idea of this thesis in the first place, and provided a lot of advice and guidance. I would also like to thank Prof. David Toman, he took part in our discussion a lot and provide us with some fantastic ideas. I wouldn't have accomplished this Master's thesis without their help. I really learned a lot from them and have improved a lot.

Besides, I would like to thank the thesis committee members: Prof. Tamer Ozsu and Prof. Richard Trefler, for their kindness in reading my thesis and giving me advice.

In addition, I would like to thank my friends, Xu Chu and Jian Li, for discussing problem in thesis with me and sharing insights on different research projects all the time.

Last but no least, I would like to thank my parents, who always loved and supported me all these years.

Contents

Chapter 1

Introduction

1.1 Overview

The “group by" operations in SQL are crucial in formulating aggregate queries over intermediate results. In particular, such operations ensure intermediate results are ordered in ways that ensure the computation of aggregate functions can be accomplished in linear time.

In this thesis, we introduce a new class of dependencies, called *clustering dependencies* (CDs), that encapsulate the necessary properties of intermediate results that ensure linear scans suffice to compute aggregate functions.

To illustrate CDs, consider an intermediate result consisting of a set of tuples that provide bindings for the following attributes: **Row ID, Order ID, Order Date, Ship Date, Ship Mode, Customer ID, Customer Name, Segment, Country, City, State, Postal Code, Region, Product ID, Category, Sub-Category, Product Name, Sales, Quantity, Discount** and **Profit**. This intermediate result comes from a table that contains orders information obtained from Superstore.

The following presents examples of CDs over this set of tuples (formal definitions will be given in the next section):

1. A CD of the form

   $$\text{Record ID} \mapsto \{\text{Order ID}\}$$

   will hold over the set of tuples iff any *list* of the same tuples that is ordered non-descending by Record ID has the property that the list is *clustered* by values of Order ID, that is, has the property that no pair of tuples in the list having the same value for Order ID will have an intervening tuple in the list with a different value of Order ID.

Record ID	Order ID
1	CA-2013-152156
2	CA-2013-152156
3	CA-2013-138688
4	US-2012-108966
5	US-2012-108966
6	CA-2011-115812
7	CA-2011-115812
8	CA-2011-115812
9	CA-2011-115812
10	CA-2011-115812
11	CA-2011-115812
12	CA-2011-115812

Figure 1.1: Dependency: Record ID $\mapsto$ {Order ID}

It is intuitive that: first, if all the records are added sequentially, then **Record ID** would represent a timestamp for each record. Records with the same **Order ID** would be grouped together because items within one order won't be separated. Second, the same **Order ID** will not be reused in the future, otherwise the logs would be corrupted. As a result, the table is guaranteed to be grouped by **Order ID**. That part of the table is shown in Figure 1.1.

2. A CD of the form

$$\text{Postal Code} \mapsto \{\text{State}\}.$$

will hold over the set of tuples iff any *list* of the same tuples that is ordered non-descending by **Postal Code** has the property that the list is *clustered* by values of **State**.

Although the fact that the table would be grouped by **State** when it is ordered by **Postal Code** is not obvious, it totally makes sense. If two places have similar postal codes, then it is very likely that they are in the same state. Moreover, it is very likely that postal codes are allocated sequentially to each state. An example is shown in Figure 1.2, we can see that postal codes within range [01040, 02740] are allocated to the state Massachusetts, meaning that any postal code in this range must not belong to any state other than Massachusetts. Also, there won't be any other postal codes from range other than [01040, 02740] that belongs to Massachusetts.

3. A CD of the form

$$\text{Product ID} \mapsto \{\text{Category, Sub-Category}\}.$$

will hold over the set of tuples iff any *list* of the same tuples that is ordered non-descending by **Product ID** has the property that the list is *clustered* by values of **Category** and **Sub-Category**.

If we look at the content of these attribute in the table as shown in Figure 1.3, we can see that **Product ID** is generated using first three letters of **Category** and **Sub-Category**. Considering the way this string (**Product ID**) is sorted: we first sort it by **Category** abbreviation (like "FUR") and then by **Sub-Category** abbreviation (like "TA"). Thus once the table is sorted on **Product ID**, it would also be sorted and of course, clustered on **Category** and **Sub-Category**.

Postal Code	State
01040	Massachusetts
01453	Massachusetts
01752	Massachusetts
01810	Massachusetts
01841	Massachusetts
01852	Massachusetts
01915	Massachusetts
02038	Massachusetts
02138	Massachusetts
02148	Massachusetts
02149	Massachusetts
02151	Massachusetts
02169	Massachusetts
02740	Massachusetts
02886	Rhode Island
02895	Rhode Island
02908	Rhode Island
02920	Rhode Island
03060	New Hampshire
03301	New Hampshire
03820	New Hampshire
04240	Maine
04401	Maine

Figure 1.2: Dependency: Postal Code $\mapsto$ {State}

Product ID	Category	Sub-Category
FUR-TA-10004607	Furniture	Tables
FUR-TA-10004619	Furniture	Tables
FUR-TA-10004767	Furniture	Tables
FUR-TA-10004915	Furniture	Tables
OFF-AP-10000026	Office Supplies	Appliances
OFF-AP-10000027	Office Supplies	Appliances
OFF-AP-10000055	Office Supplies	Appliances
OFF-AP-10000124	Office Supplies	Appliances
OFF-AP-10000159	Office Supplies	Appliances

Figure 1.3: Dependency: Product ID $\mapsto$ {Category, Sub-Category}

From the three examples above, we can see that clustering dependencies are everywhere in our daily lives and these dependencies can show interesting facts and provide insights about the data we study.

We believe Clustering Dependencies would be useful in the following four applications:

1. **Query optimization**. Integrity constraints are very helpful when it comes to query optimization. Given a query, we will typically have many potential plans from which an optimizer might choose so as to provide the correct result. However, the speed of these plans can vary enormously. Integrity constraints can be applied over these plans to improve the running speed of the query.

   As an integrity constraint, clustering dependencies can be used to optimize queries that contains **GROUP BY** operations. If a clustering dependency is already implied in the intermediate query processing steps, then the necessity of the **GROUP BY** operation can be eliminated.

   Consider the following SQL query for the previous example:

SELECT COUNT(Postcal Code)

FROM TABLE T

GROUP BY State

Normally what the query processor would do is to conduct a GROUP BY operation on table T, get the total number of postal codes for each state, and return the result. However, if the table is sorted on **Postal Code** already and we know clustering dependency: Postal Code $\mapsto$ {State} is valid, it would be no longer necessary for the query processor to conduct that GROUP BY operation.

2. **Data visualization**. We are living in a world where data is grow exponentially. But not every data owner has a comprehensive understanding of the data they own. Data visualization, on the other hand, can help people see and understand their data by providing visual rendering of the them. It has become more and more important in different fields, in particular, the business intelligence (BI) field. Visualizations help people see things that were not obvious to them before. Even when data volumes are very large, patterns can be spotted quickly and easily. Visualizations convey information in a universal manner and make it simple to share ideas with others. One of the most important benefits of visualization is that it allows us access to huge amounts of data in ways that would not be otherwise possible. There are thousands of examples of visualizations of big data, from fun and beautiful to current and historic, to financial and political. The knowledge encompassed in these various data sets would be nearly inaccessible to the casual, or even moderately interested viewer, if it was not visualized. But a good visualization gives us access to that knowledge, and does it quickly and effectively. [20]

   We use a commercial data visualization tool called Tableau [18] to demonstrate how data visualization works: most of the time, we can drag attributes to the “Rows" and “Column" section and the software will automatically generate a visual presentation of the data. For example, if we want to check our order table to see how many distinct orders there are for cities in different states. We will drag **State** and **City** to the row and *Sum of Order ID* to the column, and get a visualized figure of the result, or even a graph rendered on a real map of the United States, as shown in Figure 1.4.

   We can see that the way data visualization works is greatly related to the operation **GROUP BY**. We also found out that the speed of these

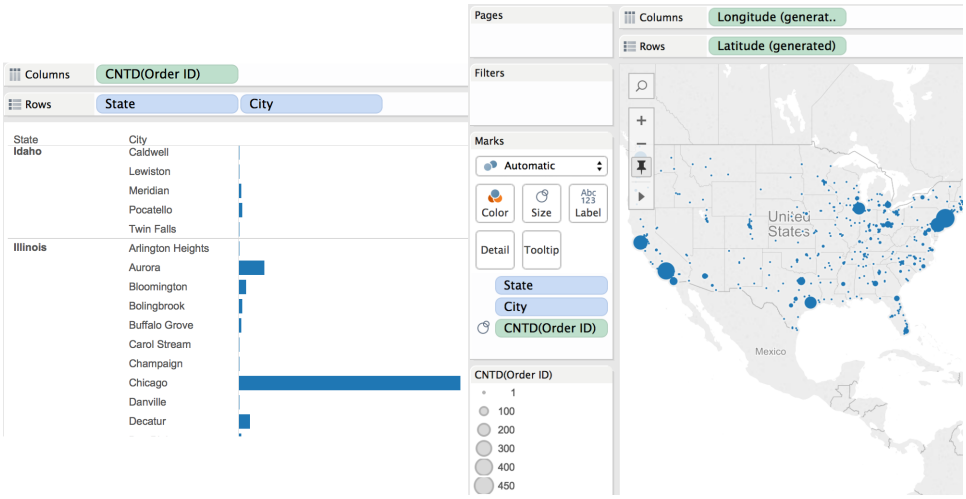

Figure 1.4: Data Visualization Example.

data visualization tools is often unsatisfactory, especially when the size of grouped by attributes is larger than 2. If we can get this operation accelerated, it would be very beneficial for data visualization.

Moreover, from the previous examples, we can observe that these dependencies not only improve the performance of the GROUP BY operation, but can also provide great insights into the data organization and the underlying semantics. This is also very helpful in data visualization because once we understand the semantics of the data, we can first load the data with much smarter strategies (the so-called smart-loading) and secondly, generate much more visual and beautiful rendering of the data.

3. **Data analysis**. Data analysis is strongly related to data visualization and is specially important and useful in the business intelligence field. The way it works is, we generate a visual representation of the data, analyze it and get the insights, and then answer questions our customers might ask about these data. Hence, once we discover the underlying semantics with clustering dependencies, we can get better much understanding of the data and thus make much better analysis and predictions.

4. **MapReduce**. MapReduce is a programming model and an associated implementation for processing and generating large data sets with a

parallel, distributed algorithm on a cluster [21][22]. It has become extremely popular in the recent decade. MapReduce works with three core phases: **Map**, **Shuffle** and **Reduce**. In the map phase, each worker (machine) will take a piece of input data and generate a key-value pair (k, v). These pairs would then be **Shuffled** to different processors according to k. Finally the reducer would process all the information associated with each k value and produce the output. An example in shown below in Figure 1.5 [23]: the input are sent to the mappers whose output are then combined and shuffled. In the shuffling and sorting stage, key-value pairs with same keys are aggregated together, which is exactly a **GROUP BY** operation. As a result, if we could discover the underlying clustering dependency that makes the keys aggregated, then we just need to a simple sort instead of all the MapReduce operations, which is much cheaper.

In this thesis, we will explore clustering dependencies from different aspects. Our major contributions are:

1. We introduced the decision problem for clustering dependencies. We did not develop a comprehensive sound and complete reasoning system. But as long as we focus more on the algorithm and application level, we believe discovering influence rules will be able help us greatly in our preceding algorithms and experiments. In addition, we also study the relation between clustering dependencies and functional/ordering dependencies. We discovered that FDs and ODs are very useful in helping us generate more useful inference rules. We presented the inference rules we discovered as well their proof.

2. We proposed a checking algorithm for clustering dependencies to check the validity of a given CD candidate. We presented the algorithm and showed that it can run with $O(NM)$ in time complexity in the worst case where N is the number of rows and M is the number of attributes on the RHS of the that CD candidate.

3. We proposed a mining algorithm for clustering dependencies. We can use this mining algorithm to discover all the potential clustering dependencies. We also showed that our inference rules would turn out to be very helpful in the pruning process.

4. We used two types of data: synthetic data and real-life data to test the performance of our mining algorithm. Synthetic data were used

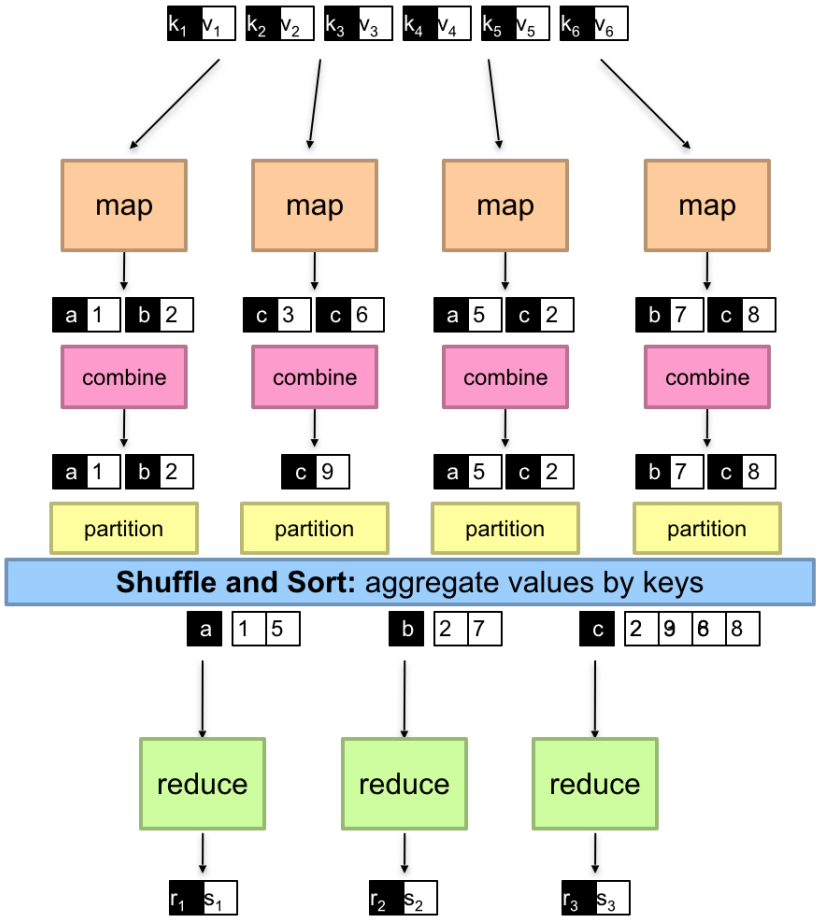

Figure 1.5: An example on MapReduce.

to test the performance of the mining algorithm from different aspects. We also used real-life data from SourceForge Research Data Archive (SRDA), a Repository of FLOSS Research Data to test the performance of our mining algorithm. The SourceForge.net web site is database driven and the supporting database includes historic and status statistics on over 320,000 projects, over 850,000 developers' activities, and over 3.4 million registered users' activities at the project management web site [15][16][17][18]. Finally we tested with the orders table referred in Chapter 1 and showed the strength of our mining algorithm.

1.2 Definitions

We have seen several intuitions of clustering dependencies from the last section. They can be represented with formal clustering dependencies as follows:

- Record ID $\mapsto$ {Order ID}.
- Postal Code $\mapsto$ {State}.
- Product ID $\mapsto$ {Category, Sub-Category}.

Definition 1.1. Clustering Dependencies (CDs)

Let R be a relation and r be one of its instances which contains a set of tuples: $t_1, t_2, ..., t_n$, let $A, B_1, B_2, ..., B_m$ be the attributes in R and let $t_i.A$ represent the value of attribute A in tuple t_i, we say r satisfies Clustering Dependency A$\mapsto\{B_1, B_2, ..., B_m\}$, iff the following first-order-logic (FOL) expression holds:

$$\begin{aligned}\forall t_x, t_y, t_z \cdot (&r(t_x) \ \wedge \ r(t_y) \ \wedge \ r(t_z) \ \wedge \\ &t_x.A \leqslant t_y.A \ \ \wedge \ \ t_y.A \leqslant t_z.A \ \ \wedge \\ &\textstyle\bigwedge_{j=1}^{m} t_x.B_j = t_z.B_j \\ &\Rightarrow \\ &\textstyle\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j)\end{aligned}$$

That is, for every three tuples in a relation set that is ordered by attribute A. If the first tuple has the exact same RHS values $(B_1, B_2, ..., B_m)$ as the third tuple, then every tuple in between (as well as the first and third tuple) should all have the same RHS values.

Recall from the last chapter that our clustering dependencies hold whenever the table is sorted by some attributes (the left-hand-side, LHS), then

it is guaranteed that the table is grouped by a list of attributes (the right-hand-side, RHS).

Not only can the LHS attribute **A** could be any one of the attributes in **R**, but we also claim it could be a self-defined attribute, possibly combined with several different attributes in **R**. We can say that we have a attribute set {***year, month, day***} from **R**, {***1990, 09, 11***}, for example. We could let our LHS be a brand-new attribute named ***date*** that is composed with the values ***year***, ***month*** and ***day*** of each tuple, and sort the table by this new attribute, a.k.a, ***1990-09-11***.

It might not be obvious why we are using an order by relationship on the LHS. Yes, we could let the LHS be a set of aggregated attributes as well just as the RHS. However, we claim that using order by on the LHS is much more useful. More importantly, according to our definition, when a set of attributes is sorted, they will be automatically aggregated as well. Moreover, with the ordering formation, we can discover a lot of more interesting dependencies. For instance, in our order table example in Chapter 1, the first two dependencies are about sorting by Record ID and Postal Code, respectively. If we just group either Record ID or Postal Code, then we will not be able to find out those two interesting dependencies.

Another potential concern might be that we only have one attribute on the LHS, despite the fact we can just merge multiple attributes into one and sort the table on that combined attribute. Theoretically, we could have multiple attributes on the LHS, and sort by these attributes, sequentially. There are two reasons for not doing this. First, consider a clustering dependency:

$$\text{Record ID} \mapsto \{\text{Order ID}\}.$$

This dependency holds with one attributes on the LHS. However we can actually add any other attributes to the LHS as secondary key, tertiary key, etc, and it won't change the validity of the dependency, because as long as the table is still sorted on Record ID as the prime key, it must always be grouped by Order ID.

Second, if we have multiple attributes on the LHS, it is very likely we will have to conduct more study on ordering dependencies in our framework which is not the focus of our thesis. We believe that would make our problem much more difficult to handle. We also claim that, for ordering dependencies, as the number of attributes grows, they would become less and less interesting and useful to us. So we will try to avoid that situation.

To make our definition more rigorous and precise, our clustering constraint applies on set of tuples rather than list of tuples. That means, the

A	B
1	2
1	2
1	3
2	4
2	4

Figure 1.6: An violation of clustering dependency.

tuples are interchangeable as long as their LHS attribute is of the same value. For example, consider the following tables in Figure 1.6. Although it is a true fact the table IS sorted by attribute A and grouped by attribute B. It, however, does not satisfy the clustering constraint $A \mapsto \{B\}$. Because we can exchange the first and the third row in the table, breaking the GROUP BY feature whilst still keeping the table sorted by A.

1.3 Comparison to Other Dependencies

Integrity constraints (ICs) have become more and more important nowadays when there is need to restrict the data values stored in a relational database with a series of constraints or rules to make the data accurate and consistent. Data integrity is the opposite of data corruption, which is a form of data loss. The overall intent of any data integrity technique is the same: ensure data is recorded exactly as intended (such as a database correctly rejecting mutually exclusive possibilities,) and upon later retrieval, ensure the data is the same as it was when it was originally recorded. In short, data integrity aims to prevent unintentional changes to information. Data integrity is not to be confused with data security, the discipline of protecting data from unauthorized parties. [19] There are a lot of integrity constraints that are already developed and well-known, like key constraints, function dependencies (FDs) [1][2] and conditional clustering constraints (CFDs) [12], ordering dependencies (ODs) [4][5][8][9], denial constraints (DCs) [14] and so on.

Definition 1.2. Functional Dependencies (FDs)[1][2]

A functional dependency states that the value of a specific attribute is uniquely determined by the values of a set of attributes. FD is a common

form of constraints in database system. Formally, when we are given a relation schema R and a relation r on R. A functional dependency $X \to A$, where $X \subseteq R$ and $A \in R$, will indicate that for any pair of tuples $t, u \in r$, if $t[x] = u[x]$ for all $x \in X$ then $t[A] = u[A]$.

The inference axioms of functional dependencies, **Armstrong's axioms**, developed by William W. Armstrong on his 1974 paper [10], can be used to infer all functional dependencies in a relational database. The axioms are **sound** in generating only functional dependencies in the closure of a set of functional dependencies (denoted as F^+) when applied to that set (denoted as F). They are also **complete** in that repeated application of these rules will generate all functional dependencies in the closure F^+.

Different approaches are introduced to deal with the mining problem of functional dependencies[2][3][6][7]. The mining methodologies can be divided into schema-driven and instance-driven approaches. TANE is a representative for the schema-driven approach [7]. It adopts a level-wise candidate generation and pruning strategy and relies on a linear algorithm for checking the validity of FDs. TANE is sensitive to the size of the schema. FASTFD is an instance-driven approach [6], which first computes agree-sets from data, then adopts a heuristic-driven depth-first search algorithm to search for covers of agree-sets. FASTFD is sensitive to the size of the instance. Both algorithms were extended in [11] for discovering CFDs [12].

Definition 1.3. Ordering Dependencies (ODs)[8][9]

An ordering dependency states that the ordering of a specific set of tuples are determined by another set of tuples. Given a relation schema R, an ordering dependency on a instance r on R is represented as $M \rightsquigarrow N$, where M and N are both sets of marked attributes, $M = \{A_1^{op_{i_1}}, A_2^{op_{i_2}}, ..., A_m^{op_{i_m}}\}$ and $N = \{B_1^{op_{j_1}}, B_2^{op_{j_2}}, ..., B_n^{op_{i_n}}\}$, where $A_1, A_2, ..., A_m$ and $B_1, B_2, ..., B_n$ are attributes from R and op can be $=, <, \leqslant, >$ or $\geqslant$. To better demonstrate this definition we need to define the operation $u[M]v$ for any two tuples u and v. We say $u[M]v$ is satisfied iff $u[A_i](op_i)v[A_i]$ is satisfied for every component $\{A_i^{op_i}\}$ in M. With that, we can say an instance I satisfies ordering dependency $M \rightsquigarrow N$ iff for any two tuples u and v, $u[M]v$ implies $u[N]v$.

Recent papers [4][5] have come up with a variation of ordering dependencies, where they use lists of attributes on both LHS and RHS rather than using sets of marked attributes. In their definition, given a relation schema R, an ordering dependency is in the form of $X \rightsquigarrow Y$, where X and Y are both

list of attributes on a relation schema R. We let $|X| = m$ and $|Y| = n$ and we will say that an ordering dependency $X \rightsquigarrow Y$ holds if, when the relation r is ordered by $x_1, x_2, ..., x_m \in X$, it would also be ordered by $y_1, y_2, ..., y_n \in Y$. In the following chapter however, we will use the first version of ordering dependency, since it would suit with our clustering dependencies better for future use.

Ordering dependencies deal with the problem of how a set of lexicographically ordered attributes are related to the ordering another set of lexicographically ordered attributes. For example if the tuples are ordered by the attributes *year, month*, then they must also be ordered by attributes *year,quarter,month* as well. With ordering dependencies implied, when queries are processed, the query optimizer can rewrite the them to achieve better performance. In this case we don't have to sort the tuples by quarter any more as long as the ordering dependency $[year, quarter, month] \rightsquigarrow [year, month]$ is implied by the relation.

The axioms of ordering dependencies has been well-studied [4][9]. While ODs can be obtained through consultation with experts, it is an expensive process and requires expertise in the constraint language at hand as well as familiarity with the current data, thus warranting the necessity of automatic mining algorithms. However, the mining algorithm for ODs is highly non-trivial. Any list of attributes can serve as LHS and RHS of an OD. Thus the space to be explored for ODs discovery is $m! \times m!$. Since OD focus on list of tuples rather than set of tuples and it has to deal with multiple attributes on both LHS and RHS, because an OD with multiple attributes on either side cannot be equivalently decomposed into smaller ODs. There have not been any efficient algorithm for OD mining yet. Even if there is, as the number of the attributes grows on both LHS and RHS, that dependency might become less and less interesting or helpful.

CFD discovery problem is also studied in [12], which not only is able to discover exact CFDs but also outputs approximate CFDs and dirty values for approximate CFDs, and in [13], which focuses on generating a near-optimal tableaux assuming an embedded FD is provided.

Denial constraints (DCs) significantly generalize FDs and CFDs. The complex form of DCs makes discovering them much harder. Chu et al. proposes an instance driven algorithm called FASTDC to discover DCs [14], which is quadratic w.r.t. the number of tuples due to the inherent complexity of checking if a DC is valid on a database instance. Two extensions, i.e., A-FASTDC, and C-FASTDC, are proposed by the same authors in order to discover DCs from dirty data, and in order to discover DCs with frequent constraints.

Chapter 2

Reasoning with Clustering Dependencies

2.1 Overview and Preliminaries

In this chapter, we consider the decision problem and inference system w.r.t our clustering dependencies. The decision problem, which contains two subproblems, the implication problem and consistency problem, is one of the basic problems in database field when it comes to integrity constraints. They can be very helpful to prune redundant dependencies and to test the validation of new dependencies and thus without doubt could greatly benefit our clustering dependency mining process.

In the following sections, we first show that the complexity of clustering dependency decision problem is at least *co-NP-complete*. Then we will introduce and prove the inference rules we have discovered, some of which are developed with the help of functional dependencies and ordering dependencies.

2.2 Decision Problem

In database theory, the decision problem is one of the most classical problems to study. There are two basic decision problems in clustering dependencies for us to study. The implication problem and the consistency problem, both defined below:

Definition 2.1. The ***implication problem*** for clustering dependencies is the question whether a specific clustering dependency d_0 be implied by a fi-

nite set of dependencies D which may contain clustering dependencies (CDs), ordering dependencies (ODs), and functional dependencies (FDs).

Definition 2.2. The ***consistency problem*** for clustering dependencies is the question whether a non-trivial model exist, given a finite set of dependencies D which may contain clustering dependencies (CDs), ordering dependencies (ODs), and functional dependencies (FDs).

We need to clarify that the consistency problem is the dual of implication problem. We will say that a set of dependencies D is inconsistent if and only if D implies a dependency of the form:

$$\forall t \cdot [r(t) \Rightarrow C]$$

where C is any unsatisfiable constraint [24].

So we will just study the implication problem instead. As mentioned before, the set of dependencies D could contain clustering dependencies (FDs), ordering dependencies (ODs) or functional dependencies (FDs). Recall the FOL definitions for each of them. In the following content, for all the FOL formulas, we will use r to represent an instance of any relational table. We use t_i to denote the tuples in the table and capitalized letters to denote attributes within the table.

Clustering Dependency: $A \mapsto \{B_1, B_2, ..., B_m\}$

$$\forall t_1, t_2, t_3 \cdot$$
$$(\ r(t_1)\ \wedge\ r(t_2)\ \wedge\ r(t_3) \wedge\ \ t_1.A \leqslant t_2.A\ \ \wedge\ \ t_2.A \leqslant t_3.A\ \ \wedge$$
$$\textstyle\bigwedge_{j=1}^{m} t_1.B_j = t_3.B_j$$
$$\Rightarrow$$
$$\textstyle\bigwedge_{j=1}^{m} t_1.B_j = t_2.B_j\ )$$

Ordering Dependency: $\{A_1^{op_1}, A_2^{op_2}, ..., A_n^{op_n}\} \rightsquigarrow \{B^{op}\}$, where

$op_i \in \{=, <, \leqslant, >, \geqslant\}$

$$\forall t_1, t_2 \cdot$$
$$(\ r(t_1) \wedge r(t_2) \wedge\ \ \textstyle\bigwedge_{j=1}^{n} t_1.A_j\ \ op_j\ \ t_2.A_j$$
$$\Rightarrow$$

$t_1.B \;\; op \;\; t_2.B \;)$

Functional Dependency: $\{A_1, A_2, ..., A_n\} \rightarrow B$

$\forall t_1, t_2 \cdot$

$(\; r(t_1) \wedge r(t_2) \wedge \quad \bigwedge_{j=1}^{n} t_1.A_j = t_2.A_j$

$\Rightarrow$

$t_1.B = t_2.B \;)$

Here we can see that FDs are special cases of ODs, hence in the following content, we will only consider ODs and CDs instead.

A study about constraint-generation dependencies [24] suggests that the implication problem for these dependencies can be linearly reduced to the validity of a universally quantified formula. The way to do this is:

First, we eliminate the quantification over tuples, which is refer as *symmetrization* in [24].

Take the simplest case of a functional dependency, $A \rightarrow B$ for example, whose logic can be written as:

$$\forall t_x, t_y \cdot (\; r(t_x) \wedge r(t_y) \wedge \quad t_x.A = t_y.A \;\Rightarrow\; t_x.B = t_y.B \;)$$

According to [24], this dependency over $r = \{t_x, t_y\}$ is equivalent to the constraint formula:

$$cf_2(d) : [t_x.A = t_y.A \Rightarrow t_x.B = t_y.B] \wedge [t_y.A = t_x.A \Rightarrow t_y.B = t_x.B]$$
$$[t_x.A = t_x.A \Rightarrow t_x.B = t_x.B] \wedge [t_y.A = t_y.A \Rightarrow t_y.B = t_y.B]$$

We can apply similar symmetrization process onto ordering dependencies and clustering dependencies as well. Such that the implication problem can be written as:

$$(\forall t_{x_1})...(\forall t_{x_3})[cf_2(OD_1) \wedge ... \wedge cf_2(OD_n) \wedge$$
$$cf_3(CD_1) \wedge ... \wedge cf_3(CD_m) \Rightarrow cf_3(CD_0)]$$

Here n and m are numbers of known ordering dependencies (ODs) and clustering dependencies (CDs). CD_0 is the clustering dependency to be implied.

We can further replace the quantification over tuples with a quantification over elements of the domain:

$$(\forall *)[cf_2(OD_1) \wedge ... \wedge cf_2(OD_n) \wedge \\ cf_3(CD_1) \wedge ... \wedge cf_3(CD_m) \Rightarrow cf_3(CD_0)] \qquad (*)$$

where $(\forall *)$ quantifies all free variables in $(*)$. The implication problem of clustering dependencies can be linearly reduced to the validity of a universally quantified formula $(*)$.

Now, take another look at our set of $[cf_2]s$ for ordering dependencies and $[cf_3]s$ for clustering dependencies, all the constraints are of the form: $(t_x.A \;\; op \;\; t_y.A)$, where $op \in \{=, \neq, <, \leqslant\}$. Notice that we didn't include $>$ and $\geqslant$ since they can be replaced with $<$ and $\leqslant$. Thus $(*)$ can be rewritten accordingly.

First we define:

$$\begin{aligned} \zeta ::= & \; t_x.A \;\; op \;\; t_y.A \\ & \mid \; \zeta_1 \wedge \zeta_2 \\ & \mid \; \zeta_1 \Rightarrow \zeta_2 \\ & \mid \; \neg\zeta \end{aligned}$$

where $op \in \{=, \neq, <, \leqslant\}$. Then we can rewrite our implication problem as:

$$(\forall *)[\; \zeta \models CD_0 \;]$$

With the constraint that ζ involves at most 3 domain variables. The paper [24] presented a theorem that states the following:

Proposition 2.3. The implication problem for clausal constraint-generating k-dependencies is:

1. *in PTIME for dependencies with one atomic $\{=, \neq, <, \leqslant\}$-constraint*
2. *co-NP-complete for dependencies with two or more atomic $\{=, \neq\}$-constraints.*
3. *co-NP-complete for dependencies with two or more atomic $\{<, \leqslant\}$-constraints.*

However, none of the cases above suits our case because we have dependencies with two or more atomic $\{=, \neq, <, \leqslant\}$-constraint. However, for the two remaining cases, they follow the fact that checking the satisfiability of a conjunction of equality and order constraints can be done in polynomial time.

These observations implies that the complexity of the implication problem for clustering dependencies is *co-NP hard*, which is unbearable. Consequently, there is a need to develop some inference rules to facilitate the implication and inference process, which could finally improve the performance of the mining algorithm that we will discuss in the next chapter.

2.3 Inference Rules

As claimed Chapter 1, functional dependencies are subsumed by ordering dependencies (The version within Definition 1.3). That is, for a functional dependency: $\{A, B\} \rightarrow C$, we could rewrite it into $\{A^{=}, B^{=}\} \rightsquigarrow \{C^{=}\}$. However, in this thesis, we are still considering them as two different dependencies. We claim that this has two benefits. First, a functional dependency is more straightforward. Second, functional dependencies are strongly related to clustering dependencies in a different way than ordering dependencies.

Inference rules can greatly help us to prone the search space. According to our study, however, it turned out that inference rules for clustering dependencies alone is not that comprehensive. However, we have been able to discover that, with the help of functional dependencies and ordering dependencies, we will find some additional rules that can help address our CD mining problem.

Once again, in the rest of this section, we will use r to represent an instance of any relational table. We use t_i to denote the tuples in the table and capitalized letters to denote attributes within the table.

We will shown these rules below:

Theorem 2.4. ***(Empty cluster):*** *For any attribute A,* $A \mapsto \{\}$.

Proof: It is a empty clustering dependency and it holds trivially. ■

Theorem 2.5. ***(Reflexivity):*** *For any attribute A,* $A \mapsto \{A\}$.

Proof: If the table is ordered by A, then it will be automatically clustered by A as well. ■

Theorem 2.6. ***(Union):*** *Given two clustering dependencies* $A \mapsto S$ *and* $A \mapsto T$ *where S and T are sets of attributes, the clustering dependency*

$A \mapsto S \cup T$ will hold. Written as:

$$\frac{A \mapsto S,\ A \mapsto T}{A \mapsto S \cup T}$$

Proof: Let $S = \{B_1, B_2, ..., B_m\}, T = \{C_1, C_2, ..., C_n\}$.
Now that we have both $A \mapsto S$ and $A \mapsto T$ being valid, then by definition we will have:

$$\forall t_x, t_y, t_z \cdot ((\ t_x.A \leqslant t_y.A \leqslant t_z.A \ \ \wedge \ \ \bigwedge_{j=1}^{m} t_x.B_j = t_z.B_j) \Rightarrow \bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j)$$

and

$$\forall t_x, t_y, t_z \cdot ((\ t_x.A \leqslant t_y.A \leqslant t_z.A \ \ \wedge \ \ \bigwedge_{j=1}^{m} t_x.C_j = t_z.C_j) \Rightarrow \bigwedge_{j=1}^{m} t_x.C_j = t_y.C_j)$$

Putting them together, we got:

$$\forall t_x, t_y, t_z \cdot ((\ t_x.A \leqslant t_y.A \leqslant t_z.A \ \ \wedge \ \bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j \ \wedge \ \bigwedge_{j=1}^{m} t_x.C_j = t_z.C_j) \Rightarrow \bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j \ \wedge \ \bigwedge_{j=1}^{m} t_x.C_j = t_y.C_j)$$

which is equivalent to the definition of clustering dependency $A \mapsto S \cup T$.

■

Theorem 2.7. ***(Expansion with FD)*** *Given a clustering dependency $A \mapsto \{B_1, B_2, ..., B_m\}$ and an **functional dependency** $S \to C$, where $S \subseteq \{B_1, B_2, ..., B_m\}$, we can infer the dependency $A \mapsto \{B_1, B_2, ..., B_m, C\}$. We can also write it as the form:*

$$\frac{A \mapsto \{B_1, B_2, ..., B_m\},\ S \to C}{A \mapsto \{B_1, B_2, ..., B_m, C\}}$$

Proof: First note that we did not specify that $m \geqslant 1$ and $S \neq \emptyset$.

If $m = 0$ or $m \neq 0$ and $S = \emptyset$, we will get $A \mapsto \{\}$ and $\{\} \to C$, with the former being a empty clustering dependency, and the latter being a functional dependency that says all the tuples have the same C value. If that is true, then we would know C is clustered as well.

For $m \geqslant 1$ and $S \neq \emptyset$, $A \mapsto \{B_1, B_2, ..., B_m\}$ gives us:

$$\forall t_x, t_y, t_z \cdot ((\ t_x.A \leqslant t_y.A \leqslant t_z.A \ \ \wedge \ \ \bigwedge_{j=1}^{m} t_x.B_j = t_z.B_j)$$

$$\overset{\Rightarrow}{\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j)}$$

Now, assume $A \mapsto \{B_1, B_2, ..., B_m, C\}$ does not hold, then by definition there is at least one triple of tuples t_x, t_y, t_z (where t_x, t_y, t_z are all in the relation r) that violates the rules, written as:

$$\begin{aligned} \exists t_x, t_y, t_z \cdot (\ & t_x.A \leqslant t_y.A \leqslant t_z.A \\ \wedge \quad & \textstyle\bigwedge_{j=1}^{m} t_x.B_j = t_z.B_j \\ \wedge \quad & t_x.C = t_z.C \\ \wedge \quad & \neg(\textstyle\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j) \vee \neg(t_x.C = t_y.C)\ ) \end{aligned} \qquad (*)$$

Now consider $(*)$, since $A \mapsto \{B_1, B_2, ..., B_m\}$ then from the above definition the left part of $(*)$ will always be false since $(\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j)$ will always be true. Now that $(\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j)$ is true, according to the functional dependency $S \to C$, we can know that $t_x.C = t_y.C$ is true as well. As a result, $(*)$ will be false which means clustering dependency $A \mapsto \{B_1, B_2, ..., B_m, C\}$ will always hold. ■

Theorem 2.8. ***(OD implication)*** *Clustering dependency can be inferred from a set of ordering dependencies, as introduced in Section 1.3. Primitively, given two ordering dependencies:* $\{A^{<}\} \rightsquigarrow \{B^{\leqslant}\}$ *and* $\{A^{=}\} \rightsquigarrow \{B^{=}\}$*, we can get clustering dependency* $A \mapsto B$*, written as:*

$$\frac{\{A^{<}\} \rightsquigarrow \{B^{\leqslant}\},\ \{A^{=}\} \rightsquigarrow \{B^{=}\}}{A \mapsto \{B\}}$$

Proof: Assume $A \mapsto \{B\}$ does not hold. Should that happen, by definition we will have at least one triple of tuples t_x, t_y, t_z (where t_x, t_y, t_z are all in the relation r) that violates the rules, written as:

$$\begin{aligned} \exists t_x, t_y, t_z \cdot (\ & t_x.A \leqslant t_y.A \leqslant t_z.A \\ \wedge \quad & t_x.B = t_z.B \\ \wedge \quad & t_x.B \neq t_y.B\ ) \end{aligned}$$

Let us divide the scenarios into four cases:

(1) $t_x.A = t_y.A = t_z.A$ (2) $t_x.A < t_y.A < t_z.A$ (3) $t_x.A < t_y.A = t_z.A$ and (4) $t_x.A = t_y.A < t_z.A$.

We analyze the four cases individually below:

Case 1: $t_x.A = t_y.A = t_z.A$, according to the ordering dependency $\{A^{=}\} \rightsquigarrow \{B^{=}\}$, $t_x.B$ should be equal to $t_y.B$.

Case 2: $t_x.A < t_y.A < t_z.A$, according to the ordering dependency $\{A^{<}\} \rightsquigarrow \{B^{\leqslant}\}$, we will know that $t_x.B \leqslant t_y.B \leqslant t_z.B$. If the violation occurs, then $t_x.B$ will be equal to $t_z.B$, which means $t_x.B = t_y.B = t_z.B$. Hence the violation would not occur.

Case 3: $t_x.A < t_y.A = t_z.A$, according to the given ordering dependencies, we can get $t_x.B \leqslant t_y.B = t_z.B$. If the violation occurs which indicates $t_x.B = t_z.B$, we will know that $t_x.B = t_y.B = t_z.B$. Hence the violation would not occur.

Case 4: $t_x.A = t_y.A < t_z.A$, according to the given ordering dependencies, we can get $t_x.B = t_y.B \leqslant t_z.B$. If the violation occurs which indicates $t_x.B = t_z.B$, we will know that $t_x.B = t_y.B = t_z.B$. Hence the violation would not occur.

Hence in general, the violation would not occur, meaning $A \mapsto \{B\}$ will always hold. ■

Theorem 2.7 also implies the following inference rules as special cases:

$$\frac{\{A^{<}\} \rightsquigarrow \{B^{<}\},\ \{A^{=}\} \rightsquigarrow \{B^{=}\}}{A \mapsto \{B\}}$$

$$\frac{\{A^{<}\} \rightsquigarrow \{B^{=}\},\ \{A^{=}\} \rightsquigarrow \{B^{=}\}}{A \mapsto \{B\}}$$

Theorem 2.9. ***(Expansion with OD)*** *Given a clustering dependency $A \mapsto \{B_1, B_2, ..., B_m\}$ and two* ***ordering dependencies*** *$\{A^{<}\} \rightsquigarrow \{C^{\leqslant}\}$, $\{A^{=}\} \rightsquigarrow \{C^{=}\}$, we can get the dependency $A \mapsto \{B_1, B_2, ..., B_m, C\}$ being true. We can also write it as the form of:*

$$\frac{A \mapsto \{B_1, B_2, ..., B_m\},\ \{A^{<}\} \rightsquigarrow \{C^{\leqslant}\},\ \{A^{=}\} \rightsquigarrow \{C^{=}\}}{A \mapsto \{B_1, B_2, ..., B_m, C\}}$$

Proof: Follows immediately from Theorem 2.5 and Theorem 2.7. ■

Theorem 2.10. *Given an ordering dependency $\{A^{<}\} \mapsto \{B_1^{<}\}$, and a bunch of functional dependencies $A \rightarrow B_1$, $A \rightarrow B_2$, ... , $A \rightarrow B_m$, we will have*

clustering dependency as $A \mapsto \{B_1, B_2, ..., B_m\}$. *As for:*

$$\frac{\{A^{<}\} \mapsto \{B_1^{<}\}, A \to B_1, A \to B_2, ..., A \to B_m}{A \mapsto \{B_1, B_2, ..., B_m\}}$$

Proof: Assume $A \mapsto \{B_1, B_2, ..., B_m\}$ does not hold, then by definition there is at least one triple of tuples that violates the rules, written as:

$$\begin{aligned} \exists t_x, t_y, t_z \cdot (\ & t_x.A \leqslant t_y.A \leqslant t_z.A \\ \wedge \quad & \textstyle\bigwedge_{j=1}^{m} t_x.B_j = t_z.B_j \\ \wedge \quad & \neg(\textstyle\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j) \end{aligned}$$

According to functional dependency $A \to B_1$ and ordering dependency $\{A^{<}\} \mapsto \{B_1^{<}\}$, we will know that

1. When A is sorted, B_1 will be sorted as well.
2. All the tuples with the same A value should have the same B_1 value. Likewise, all the tuples with the same B_1 value should have the same A value.

Now, for a violation t triple (t_x, t_y, t_z), we must a have $\bigwedge_{j=1}^{m} t_x.B_j = t_z.B_j$, and thus we will know $t_x.B_1 = t_z.B_1$. According to (2), we can get $t_x.A = t_z.A$, furthermore, $t_x.A = t_y.A = t_z.A$.

At this point, With $t_x.A = t_y.A$ and all the remaining functional dependencies $A \to B_2, A \to B_3, ..., A \to B_m$. We will know that $(\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j)$. Should that be true, the violation would never occur. Thus the original clustering constraint will hold. ■

To summarize, we have discovered 7 inference rules for clustering dependencies, shown in Table 2.1 .

Type	Inference Rule
Empty Cluster	$A \mapsto \{\}$
Reflexivity	$A \mapsto \{A\}$
Union	$\dfrac{A \mapsto S,\ A \mapsto T}{A \mapsto S \cup T}$
Expansion with FD	$\dfrac{A \mapsto \{B_1, B_2, ..., B_m\},\ S \to C}{A \mapsto \{B_1, B_2, ..., B_m, C\}}$
OD implication	$\dfrac{\{A^{<}\} \rightsquigarrow \{B^{\leqslant}\},\ \{A^{=}\} \rightsquigarrow \{B^{=}\}}{A \mapsto \{B\}}$
Expansion with OD	$\dfrac{A \mapsto \{B_1, B_2, ..., B_m\},\ \{A^{<}\} \rightsquigarrow \{C^{\leqslant}\},\ \{A^{=}\} \rightsquigarrow \{C^{=}\}}{A \mapsto \{B_1, B_2, ..., B_m, C\}}$
Combination	$\dfrac{\{A^{<}\} \mapsto \{B_1^{<}\}, A \to B_1, A \to B_2, ..., A \to B_m}{A \mapsto \{B_1, B_2, ..., B_m\}}$

Table 2.1: Influence rules for clustering dependencies.

Chapter 3

Checking and Mining Algorithms for Clustering Dependencies

3.1 Clustering Dependency Validity Checking Algorithm

3.1.1 Problem Definition

In this chapter, we will introduce clustering dependency validity checking algorithm.

Given a relational instance r, we use t_i to represent the tuples in r and upper-case letters $A, B_1, B_2, ..., B_k$ to represent attributes in r. Now given a potential clustering dependency on that instance, we want to determine whether A$\mapsto\{B_1, B_2, ..., B_m\}$ holds in r.

Recall the FOL representation of clustering dependencies presented in Definition 1.2:

$$\forall t_x, t_y, t_z \cdot (r(t_x) \wedge r(t_y) \wedge r(t_z) \wedge (t_x.A \leqslant t_y.A) \wedge (t_y.A \leqslant t_z.A)$$
$$\wedge(\bigwedge_{i=1}^{m} t_x.B_i = t_z.B_i) \Rightarrow (\bigwedge_{i=1}^{m} t_x.B_i = t_y.B_i))$$

a brute-force checking algorithm is quite obvious: we could enumerate every order triplet of tuple in the form of t_1, t_2, t_3 and verify if they satisfy the above FOL expression. If every triple satisfies the expression, we will say that this clustering dependency holds.

The correctness of this algorithm is obvious because if no violation occurs during the algorithm then we can assure that all the tuples satisfy the definition of CD and thus that potential CD must be valid. However, the brutal-force algorithm would take $O(N^3)$ time to enumerate all the triples and yet another $O(m)$ time for to compare each attribute, making the time complexity for that would raising to $O(N^3m)$ where N is the number of tuples in instance r, and m is the number of attributes on the RHS. We can see that as N grows, the time cost of this algorithm would be huge.

In the following content, we will present an algorithm that gives us the time complexity of $O(Nm)$, where N is the number of tuples and m is the size of RHS attributes of the clustering dependency to be verified.

3.1.2 Introduction of the Algorithm

Let us first consider the case where there is only one LHS attribute. We assume all the clustering dependencies to check are of the form:

$$A \mapsto \{B_1, B_2, ..., B_m\}.$$

In the following content, we will denote our instance with t, and use t_i to represent each tuple in the instance.

Before we introduce the algorithm. We will define another important and practical concept: **"Interesting" Clustering Dependencies**.

Definition 3.1. Interesting Clustering Dependency

Let $A \mapsto \{B_1, B_2, ..., B_m\}$ be a clustering dependency. We say this clustering dependency is **interesting** if either of the following two cases holds:

1. $B_1, B_2, ..., B_m$ are not keys, and their values are not identical on every row.

2. Both A and $B_1, B_2, ..., B_m$ are sorted keys.

Consider the first case, the reason we add this constraint is, as the number of attributes m grows, the value combination of $B_1, B_2, ..., B_m$ will be more likely to be different from each other, and eventually could become key of the table. Although by definition they are still valid clustering dependencies, they can provide us with no useful information and are not worth studying. Hence we should consider them as "uninteresting" clustering dependencies.

Also, if the RHS values of a CD are identical, that CD will hold trivially but we are not interested in that case.

The only exception for that is the second case, when the LHS is also a key and both sides are sorted. This would a special case of an ordering dependency. Hence, we could consider them “interesting" as well.

That being said, before our checking algorithm, it is necessary to check whether this CD candidate is interesting. The checking for attributes that contain identical values can be done when the table is read. The process of checking RHS for keys will be done prior to the checking algorithm for that CD candidate.

The details of the algorithm is shown below:

Preprocessing:

1. After reading in a table, found out the attributes that have only one identical value, and remove this attribute from the input.
2. For each RHS candidate, check if this RHS forms a key. If it does and the LHS is not, we will not proceed with this CD candidate.

Input for the checking algorithm:

- a table T with N rows and M columns, that contains the tuples $\{t_1, t_2, ..., t_N\}$
- one LHS attribute, marked as A
- a set of RHS attributes, marked as $B_1, B_2, ..., B_m$, where $m \leqslant M$

Main algorithm:

During the main part of the algorithm, we do a linear scan of all the tuples. Our algorithm terminates as soon as a violation is found. If the algorithm performed a scan of all tuples without finding a violation, we could consider this potential clustering dependency as valid. Our algorithm ensures that, when it reaches tuple t_i, then all the tuples before that, i.e., t_1 to t_{i-1}, will satisfy that clustering dependency. Consequently, when we are at t_i, our job is to verify that adding this tuple will not cause any violation.

To help explaining the algorithm we first introduce some variables and definitions that are used in the algorithm.

- **LHS-cluster** An LHS-cluster will represent a group of tuples with the same LHS value. Since for the rest of the algorithm the tuples are considered to be sorted by the LHS attribute, tuples with the same LHS value must belong to the same (and that only) LHS-cluster. Moreover, when we say "current LHS-cluster", we mean the group of tuples that has the same A value as tuple t_{i-1} (since we will be verifying t_i at that moment), and of course, these tuples must be adjacent to one another.

- **RHS-value** A RHS-value indicates the value combination of all the RHS attributes in that clustering dependency candidate, a.k.a, $B_1, B_2, ..., B_m$. We would say that two tuples have the same RHS-value iff these two tuples have the same B_i for every $i \in \{1, 2, .., m\}$.

- **currentClusterMustIdentical** currentClusterMustIdentical is a boolean variable that could loosely be translated into "the tuples in the current LHS-cluster must have identical RHS-value". If this variable is true, any new tuple that comes into the current LHS-cluster with a different RHS-value is considered as a violation. This variable is used to check for violation when current tuple t_i is in same LHS-cluster as t_{i-1}.

- **currentClusterIdentical** currentClusterIdentical is another boolean variable that could be translated into "the tuples in the previous LHS-cluster have identical RHS-value". This variable is used to check for violation when current tuple t_i is in different LHS-cluster as t_{i-1}.

Definition 3.2. Valid Tuple/Row
Let A$\mapsto\{B_1, B_2, ..., B_m\}$ be a CD candidate. Our algorithm will perform a linear scan from the first tuple to the last tuple. In this process, we say a tuple t_i is **valid** (i indicates the sequential number of that tuple) iff

1. Tuples $t_1, t_2, ..., t_{i-1}$ are valid.

2. Within an instance composed with tuples $\{t_1, t_2, ..., t_i\}$, A$\mapsto\{B_1, B_2, ..., B_m\}$ holds.

Intuitively, we say a tuple is valid if adding this tuple will not compromise the given CD.

The main logic of the algorithm is shown in Algorithm 1. We first check if RHS is a key and sort the table by the LHS attribute A. We choose to use a hash table data structure to save all the RHS-value combinations. We will use this to check whether that RHS-value has appeared in previous tuples.

The time complexity for each insert and search operation in a hash table is amortized O(1). Consequently, **the total time complexity for CD checking algorithm would be O(Nm)**, since we need $O(N)$ time to go through all the tuples and yet another $O(m)$ time to check all the attributes on the RHS of each tuple in the worst case.

For each row, we will check its validity based on four cases.

1. Different LHS-cluster, RHS-value doesn't exist.

   The RHS-value of current tuple t_i is not stored in the hash table and is not in the same LHS-cluster as t_{i-1}.

2. Same LHS-cluster, RHS-value doesn't exist.

   The RHS-value of current tuple t_i is not stored in the hash table and is in the same LHS-cluster as t_{i-1}.

3. Different LHS-cluster, RHS-value exists.

   The RHS-value of current tuple t_i is already stored in hash table and is not in the same LHS-cluster as t_{i-1}.

4. Same LHS-cluster, RHS-value exists.

   The RHS-value of current tuple t_i is already stored in hash table and is in the same LHS-cluster as t_{i-1}.

If any violation occurs, the algorithm will terminate. We will discuss the outcomes for each case in the next section. Finally, if the algorithm scanned all the tuple without finding a violation, we would consider this CD candidate as a valid CD.

3.1.3 Correctness Proof

As introduced in the last section, the four situations would cover every case for each tuple. Hence every incoming tuple must fall into one of the four categories. In the algorithm, we will conduct 4 different types of **condition check**, one for each case, respectively, to determine if t_i is violation. The outcome for each situation and the corresponding proof are shown below:

Case 1: Different LHS-cluster, RHS-value doesn't exist. In this case, t_i would always be valid.

Consider the FOL for CD:

$\forall t_x, t_y, t_z \cdot (r(t_x) \wedge r(t_y) \wedge r(t_z) \wedge (t_x.A \leqslant t_y.A) \wedge (t_y.A \leqslant t_z.A) \wedge (\bigwedge_{j=1}^{m} t_x.B_j = t_z.B_j) \Rightarrow (\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j))$.

Data: input table and a potential dependency A$\mapsto\{B_1, B_2, ..., B_m\}$,
number of rows N, number of columns M
Result: True or False(whether the above dependency holds)
if (RHS is key) return false; `// uninteresting CD`
if (N==1) return true; `// if only one tuple`
sortTuplesBy(table, A); `// sort the tuples by LHS attribute`
HashInsert($r_1.B_{1..m}$); `// add the first tuple into hash table`
currentClusterIdentical = true;
currentClusterMustIdentical = false;
for $i \leftarrow 2$ **to** l **do**
  if *HashTableFind($r_i.B_{1..m}$)==false AND* $r_i.A \neq r_{i-1}.A$ **then**
    `// CASE 1:`
    currentClusterIdentical=true;
    currentClusterMustIdentical=false;
    HashTableInsert($r_i.B_{1..m}$);
    continue ; `// passes`
  else if *HashTableFind($r_i.B_{1..m}$)==false AND* $r_i.A = r_{i-1}.A$ **then**
    `// CASE 2:`
    if *currentClusterMustIdentical == false* **then**
      currentClusterIdentical = false;
      HashTableInsert($r_i.B_{1..m}$);
      continue ; `// passes`
  else if *HashTableFind($r_i.B_{1..m}$)==true AND* $r_i.A \neq r_{i-1}.A$ **then**
    `// CASE 3:`
    if *currentClusterIdentical==true AND* $r_i.B_{1..m} = r_{i-1}.B_{1..m}$ **then**
      currentClusterMustIdentical = true;
      currentClusterIdentical = true;
      continue ; `// passes`
  else if *HashTableFind($r_i.B_{1..m}$)==true AND* $r_i.A = r_{i-1}.A$ **then**
    `// CASE 4:`
    if *currentClusterIdentical==true AND* $r_i.B_{1..m} = r_{i-1}.B_{1..m}$ **then**
      currentClusterMustIdentical = true;
      currentClusterIdentical = true;
      continue ; `// passes`
  `// If we got here, an violation occurred`
  return false;
end

Algorithm 1: CD checking algorithm

When we are at t_i, all the previous tuples satisfy the clustering dependency. That is, for all $x, y, z < i$, the above formula must hold. To prove adding t_i will not break anything, we could replace one of the variables with t_i. Since t_i has different LHS-value and RHS-value from any previous tuples, we will have to prove the following:

$\forall t_x, t_y, t_i \cdot (r(t_x) \wedge r(t_y) \wedge r(t_i) \wedge (t_x.A \leqslant t_y.A) \wedge (t_y.A < t_i.A) \wedge (\bigwedge_{j=1}^{m} t_x.B_j = t_i.B_j) \Rightarrow (\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j))$.

The LHS of the this FOL expression could never be true. Thus in case 1, t_i could never be a violation. ■

Case 2: Same LHS-cluster, RHS-value doesn't exist. In this case, t_i would be valid iff "currentClusterMustIdentical" is false.

" $\Rightarrow$ " :

Since t_i is in the same LHS-cluster with t_{i-1}, and that it has a new RHS-value, there must be at least two different RHS-values in the current LHS-cluster. Hence "currentClusterMustIdentical" could not be true.

" $\Leftarrow$ " :

Since t_i has different RHS-value from all the previous tuples, it will not effect the previous tuples with different LHS-value, because its RHS-value could not equal to any one of those. Hence we only need to investigate the tuples in the current LHS-cluster. We can see that for any three tuples t_x, t_y in this LHS-cluster, t_i would be valid if we have:

$\forall t_x, t_y, t_i \cdot (r(t_x) \wedge r(t_y) \wedge r(t_i) \wedge (t_x.A \leqslant t_y.A) \wedge (t_y.A \leqslant t_i.A) \wedge (\bigwedge_{j=1}^{m} t_x.B_j = t_i.B_j) \Rightarrow (\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j))$.

Since t_x, t_y and t_i are all in the same LHS-cluster, they should have the same LHS value:A. Hence we can rewrite the formula as:

$\forall t_x, t_y, t_i \cdot (r(t_x) \wedge r(t_y) \wedge r(t_i) \wedge (t_x.A = t_y.A) \wedge (t_y.A = t_i.A) \wedge (\bigwedge_{j=1}^{m} t_x.B_j = t_i.B_j) \Rightarrow (\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j))$.

Now that "currentClusterMustIdentical" is false, $(\bigwedge_{j=1}^{m} t_x.B_j = t_i.B_j)$ would never be true. It makes the LHS of this FOL false, and the whole FOL expression true. Thus t_i would be valid. ■

Case 3: Different LHS-cluster, RHS-value exists. In this case, t_i would be valid iff (1) "currentClusterIdentical" is true and (2)$\mathbf{t_i}$ has same RHS-value as $\mathbf{t_{i-1}}$.

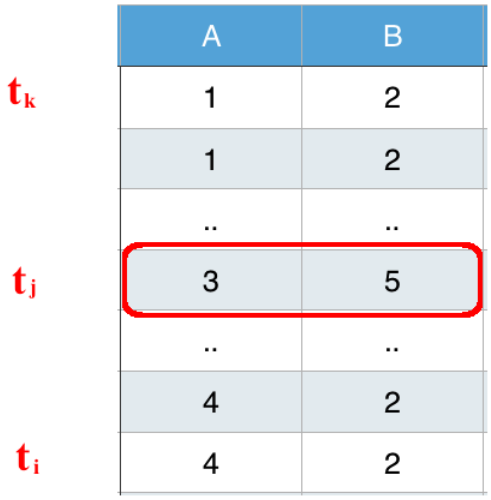

Figure 3.1: Violation: tuple t_j

" $\Rightarrow$ " :

Now that we have a tuple t_i whose RHS-value has occurred before, and it is in a different LHS-cluster as t_{i-1} and all the previous tuples. Consider any tuple that has the same RHS-value as t_i, say t_k, then we can know the following two facts are true:

i) For any t_j where $k \leqslant j \leqslant i$, it should have the same RHS-value as t_k.

ii) For any t_j that are in the same LHS-cluster as t_k, it should have the same RHS-value as t_k.

Fact i) can be easily seen. Since t_i and t_k have the same RHS-value, any tuple between them should also have the same RHS-value. Otherwise the RHS attributes would not be clustered. For example, in Figure 2.1, t_j does not have the same RHS-value as t_k and thus will corrupt the CD regulation.

Fact ii) states that any tuple in the same LHS-cluster as t_k should also have the same RHS-value as t_k. This indicates that if there is any tuple in any cluster that has the same RHS-value as t_i, that LHS-cluster should have identical RHS-value. If we look at Figure 2.2, t_k and t_j are in the same LHS-cluster but have different RHS-values. Now if we swap t_j and t_k, t_j would be sitting between t_i and t_k with different RHS-value, which is a violation of CD.

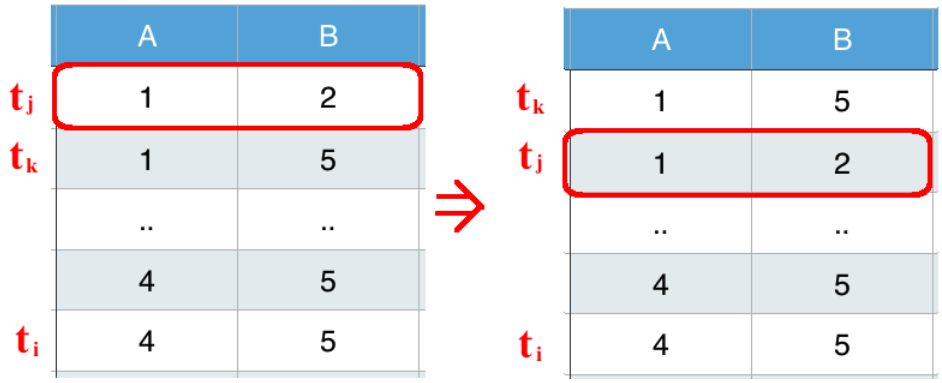

Figure 3.2: Violation: tuple t_j

From the 2 facts above we can conclude that the LHS-cluster before t_i must have identical RHS-value, and "currentClusterIdentical" would then true. Also, since the RHS-value already exists, then there must be at least one tuple that has the same RHS-value as t_i, and since those tuples must be adjacent, we can know that t_{i-1} must have the same RHS-value as t_i.

" $\Leftarrow$ " :

Again, consider the FOL expression:

$\forall t_x, t_y, t_i \cdot (r(t_x) \wedge r(t_y) \wedge r(t_i) \wedge (t_x.A \leqslant t_y.A) \wedge (t_y.A < t_i.A) \wedge (\bigwedge_{j=1}^{m} t_x.B_j = t_i.B_j) \Rightarrow (\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j))$.

First consider the cases where t_x is in the same LHS-cluster as t_{i-1}, we will have:

$\forall t_x, t_y, t_i \cdot (r(t_x) \wedge r(t_y) \wedge r(t_i) \wedge (t_x.A = t_y.A) \wedge (t_y.A < t_i.A) \wedge (\bigwedge_{j=1}^{m} t_x.B_j = t_i.B_j) \Rightarrow (\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j))$.

Since "currentClusterIdentical" is true and t_i has same RHS-value as t_{i-1}, it indicates that t_i has the same RHS-value as every tuple in the previous LHS-cluster. Hence the expression holds.

On the other hand, if t_i is not in the same LHS-cluster as t_{i-1}. Since "currentClusterIdentical" is true and t_i has same RHS-value as t_{i-1}, we would know that all the previous LHS-cluster as well as t_i share one common RHS-value. In the expression:

$\forall t_x, t_y, t_i \cdot (r(t_x) \wedge r(t_y) \wedge r(t_i) \wedge (t_x.A \leqslant t_y.A) \wedge (t_y.A < t_i.A) \wedge (\bigwedge_{j=1}^{m} t_x.B_j = t_i.B_j) \Rightarrow (\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j))$.

For the LHS to be true, t_x should have the same RHS-value as t_i. Also, we already know that tuples until t_{i-1} are all valid, which indicates that t_x and its LHS-cluster also share that common RHS-value with t_i and the previous LHS-cluster. Thus the formula must be true. ■

Case 4: Same LHS-cluster, RHS-value exists. In this case, t_i is valid iff (1) "currentClusterIdentical" is true and (2)t_i has same RHS-value as t_{i-1}.

" $\Rightarrow$ " :

Using proof by contradiction, assume the statement is false, then either (1) or (2) would be false.

Consider (1) when the existed RHS-value belongs to a tuple in the current LHS-cluster. Let that tuple be t_j. Since (1) is false, there would be at least one tuple that has different RHS-value as t_i, say t_k. We could see that this violates the definition of clustering dependencies, because we can swap the order of the tuples, make t_k sit between t_i and t_j and thus make a violation.

If the existed RHS-value belongs to a tuple in a LHS-cluster other than the current one, say t_j. Since (1) is false, there would be at least one tuple that has different RHS-value as t_i, say t_k. Again, we can swap the order of the tuples and make t_k lay between t_i and t_j, which would cause a violation of CD.

Consider (2), t_i has a RHS-value different than t_{i-1}. We could know that there exists one tuple t_k whose RHS-value equals to t_i, and that could not be t_{i-1}. Hence no matter where it is, there would be no valid CD within the table because t_{i-1} sits in-between t_i and t_k. An example is shown in Figure 3.3.

" $\Leftarrow$ " :

Consider the FOL expression for a CD:

$\forall t_x, t_y, t_z \cdot (r(t_x) \wedge r(t_y) \wedge r(t_z) \wedge (t_x.A \leqslant t_y.A) \wedge (t_y.A \leqslant t_z.A) \wedge (\bigwedge_{j=1}^{m} t_x.B_j = t_z.B_j) \Rightarrow (\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j))$.

Assume that t_y lies in the same LHS-cluster of t_i, then we need to prove:

$\forall t_x, t_y, t_i \cdot (r(t_x) \wedge r(t_y) \wedge r(t_i) \wedge (t_x.A \leqslant t_y.A) \wedge (t_y.A = t_i.A) \wedge (\bigwedge_{j=1}^{m} t_x.B_j = t_i.B_j) \Rightarrow (\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j))$.

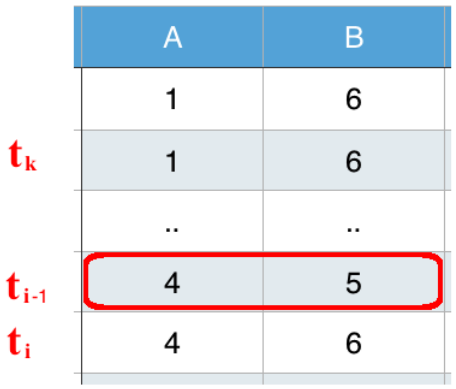

Figure 3.3: A violation of CD

Since “currentClusterIdentical" is true, t_y and t_i must have the same RHS-value. If the LHS of the FOL expression is false, the FOL expression itself would then be true. Otherwise, if the LHS of the FOL expression is true, it would indicate that t_x has same RHS-value as t_i. From that we can know t_i, t_y and t_i have the same RHS-value, in which case the RHS of the FOL expression would be true.

On the other hand, if t_y is not in the previous LHS-cluster of t_i, for the LHS of the FOL expression to be true, tuples between t_x and t_i (inclusive) must share common RHS-value. In this case, t_x and its corresponding LHS-cluster must also share that common RHS-value. The expression:

$\forall t_x, t_y, t_z \cdot (r(t_x) \wedge r(t_y) \wedge r(t_z) \wedge (t_x.A = t_y.A) \wedge (t_y.A = t_z.A) \wedge (\bigwedge_{j=1}^{m} t_x.B_j = t_z.B_j) \Rightarrow (\bigwedge_{j=1}^{m} t_x.B_j = t_y.B_j))$

would hold trivially since t_x, t_y and t_z will then share common RHS-value. ■

3.2 Clustering Dependency Mining Algorithm

In this section, we will study the mining problem for clustering dependencies. Given a table or a set of dataset, we would like to know if there are any underlying interesting clustering dependencies that could let us understand our data and make better use of them.

However as the size of the table grows, and in particular, when the number of columns grows, the time cost of the mining algorithm could be a

big issue for us. Thus we will have to find ways to optimize our algorithm and prune the search space. We will talk about it in the following content.

3.2.1 Problem Definition

Given a relation instance $r \in R$, where $R = \{R_0, R_1, ..., R_k\}$, a set of tuples, our CD mining algorithm is aiming to find out every clustering dependency within R. These dependencies will be of the form $R_i \rightarrow S$, where $S = \{R_{j_1}, R_{j_2}, ..., R_{j_l}\}$, and $1 \leqslant l \leqslant k + 1$.

We have two more specifications to make:

- First, to simply the problem, we will assume the LHS attribute of the mining algorithm is known beforehand in the following content. We do this because whichever attribute we use as the LHS, it will not make any difference to the algorithm.
- Second, for the LHS attribute R_i, $R_i \notin S$, simply because $R_i \rightarrow \{R_i\}$ is trivial.

3.2.2 Algorithm Introduction

As claimed before, our problem is to find all the possible clustering dependencies in the form of $R_0 \rightarrow S$ within the S lattice, suppose S={A,B,C,D}, then our lattice will look like this in Figure 3.4:

As claimed before, we will consider that the LHS attribute is already known to the algorithm. We will let it be R.

Definition 3.3. A **Node in the Lattice** does not mean the set of attributes represented in the lattice cell, it refers to the corresponding clustering dependency whose RHS equals the set of attributes in that cell.

Definition 3.4. Validity of Node
The **validity** of a node indicates whether the clustering dependency represented by that node is valid.

As an intuition, consider one of the inference rules: the union rule:

$$\frac{A \mapsto \{B_1, B_2, ..., B_m\}, A \mapsto C}{A \mapsto \{B_1, B_2, ..., B_m, C\}}$$

This inference rule can be used two ways in CD mining algorithm:

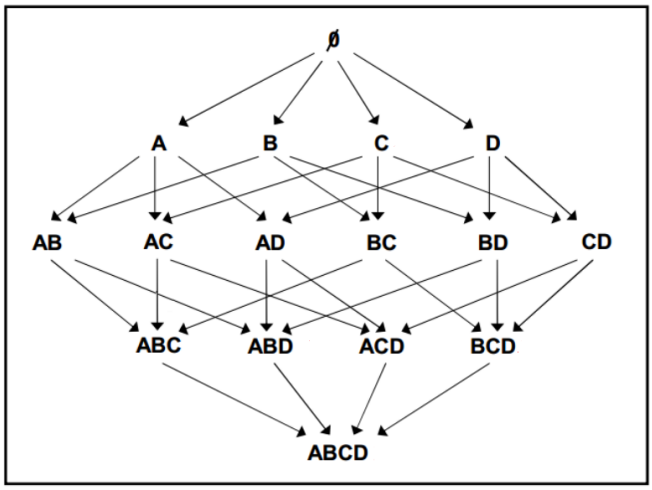

Figure 3.4: An example of the complete lattice with S={A,B,C,D}.

- The first thing is that if both $A \mapsto \{B_1, B_2, ..., B_m\}$ and $A \mapsto C$ are true, we will know that $A \mapsto \{B_1, B_2, ..., B_m, C\}$ is true as well.
- Another way to look at this is, if $A \mapsto \{B_1, B_2, ..., B_m, C\}$ is false, and one of the premises, say, $A \mapsto \{B_1, B_2, ..., B_m\}$ is true, then the other premise must be false.

With that being said, once we acquire the validity of some nodes in the lattice, we will be able to discover all of the clustering dependencies in the lattice. The mining process can be done in two ways: a top-down approach and a bottom-up approach. Of course we could apply these two methods at the same time in our algorithm.

In an ideal world, with all the inference rules we know, we would be able to discover all of the clustering dependencies. But that is hardly the case most of the time. We are very likely to be stuck at some node where there is no more inference rules to help us go forward. Should this happen, we will need to use the checking algorithm we proposed in the last chapter to verify the clustering dependencies whose validity cannot be deduced solely from inference rules.

The way we use checking algorithm within mining algorithm is, we will first explore CDs with inference rules, until at some point no more dependencies can be inferred. At that moment, we will use the checking algorithm to check the validity of a CD, for the mining algorithm to proceed.

In the next section, we will see in detail how inference rules can be used to mine clustering dependencies.

3.2.3 Algorithm Implementation

Before we introduce the main part of the algorithm, there is another clarification we need to make.

Since we are only concerned with the mining algorithm for clustering dependencies, we will consider that **all the ordering dependencies and functional dependencies are already known to the algorithm**. That is, they are considered to be part of the input to the mining algorithm. We will first discover all the FDs and ODs and feed the result to the CD mining algorithm. To this end, we need to following definition:

Definition 3.5. An **Influence Edge** is an directed edge between two nodes in the lattice. There will be an influence edge from node N_1 to node N_2 iff the validity of N_2 can be inferred by N_1 and existing FDs/ODs.

Now we will see how we will use different types of inference rules to optimize the CD mining algorithm.

Case 1. Consider the OD implication rule:

$$\frac{\{A^{<}\} \rightsquigarrow \{B^{\leqslant}\},\ \{A^{=}\} \rightsquigarrow \{B^{=}\}}{A \mapsto \{B\}}$$

We can directly obtain some clustering dependencies from ordering dependencies, using Theorem 2.7:

This theorem can be easily applied in the algorithm. We only need to go through every OD and see if any CD that can be inferred from them.

Case 2. Consider the union rule in Theorem 2.5 with an example shown below:

$$\frac{A \mapsto \{B\},\ A \mapsto \{C, D\}}{A \mapsto \{B, C, D\}}$$

As introduced earlier, this inference rule can be used from both directions:

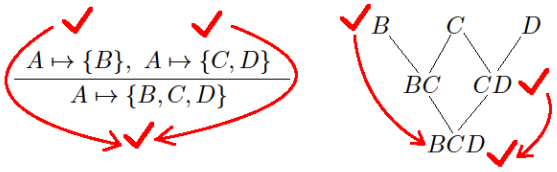

Figure 3.5: Validate new clustering dependency with union rule (top-down).

For the top-down approach, assume we have already identified that both $A \mapsto \{B\}$ and $A \mapsto \{C, D\}$ are true. Based on Theorem 2.5, we can know that $A \mapsto \{B, C, D\}$ must hold, as shown in Figure 3.5.

In this situation, once we identified a valid clustering dependency (say, $A \mapsto \{B\}$), we will visted all the other unchecked nodes in the lattice that are disjoint with attributes in current RHS, and mark their union ($\{B, C, D\}$) as a valid node. We could then get a new valid CD: $A \mapsto \{B, C, D\}$.

As for the bottom-up approach, if we know that $A \mapsto \{B, C, D\}$ is false, and $A \mapsto \{B\}$ is valid, we can know that $A \mapsto \{C, D\}$ must be false.

That is, once we have identified some invalid clustering dependency $R_0 \mapsto S$, we can check for all the dependencies whose RHS is subset of the RHS of current dependency $R_0 \mapsto S_1$. For example in our case, the RHS of $A \mapsto \{B\}$ is a subset of the RHS of $A \mapsto \{B, C, D\}$. Once we find this, we can mark $R_0 \mapsto S_2$ as invalid clustering dependency where $S_2 = S - S_1$ (in our case, $A \mapsto \{C, D\}$), as shown in Figure 3.6.

Case 3. Consider the expanding rules with FD/OD with an example shown below:

$$\frac{R \mapsto \{B\},\ B \to C, D}{R \mapsto \{B, C, D\}}$$

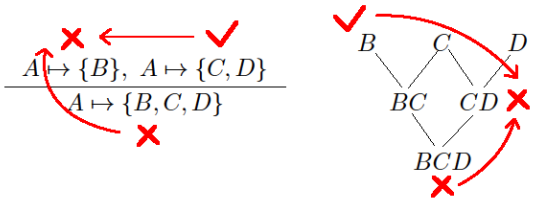

Figure 3.6: Invalidate new clustering dependency with union rule (bottom-up).

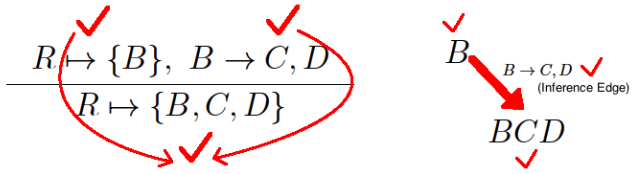

Figure 3.7: Validate new clustering dependency with FD/OD (top-down).

As claimed at the beginning of the section, we would assume that the FDs and ODs are already known. We can then account for them as base knowledge and represent them as **Inference Edges** in the lattice.

For example, if we know a clustering dependency $R \mapsto \{B\}$ and a functional dependency $B \to C, D$ (which is a combination of $B \to C$ and $B \to D$):

Based on Theorem 2.6, we will assume that clustering dependency $R \mapsto \{B, C, D\}$ holds. In this case, $B \to C, D$ will act as an **Inference Edges** that connects nodes B and BCD. It is not an edge in the original lattice, as we can see in Figure 3.7.

Likewise, there are two different situations we need to consider here,

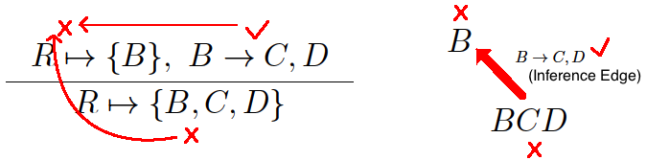

Figure 3.8: Invalidate new clustering dependency with FD/OD (bottom-up).

which corresponds to two different types of edges, one pointing downwards in the lattice, as we just saw, and the other pointing upwards.

In the first situation, we have one valid clustering dependency and one given FD/OD. We will be able to validate a new clustering dependency from them. In the algorithm, when we are at a valid node in the lattice, if there is an influence edge representing FD/OD pointing down, then we can just mark the node on the other side of the edge as valid.

The other situation is as follows: we are an invalid clustering dependency and a related FD/OD. Based on Theorem 2.6, we can know that one of the premises must be wrong in this case, and since that given FD/OD must be correct, the clustering dependency premise should be wrong. In this case, we will draw an influence edge pointing up from the invalid node to the node representing that CD premise.

For example, if we know that $R \mapsto \{B, C, D\}$ is an invalid CD, and we know an FD $B \to CD$. Then following the inference edge, we can deduce that the clustering dependency $R \mapsto \{B\}$ is invalid as well, as shown in Figure 3.8.

3.2.4 Main Algorithm

As introduced before, we will use both top-down and bottom-up search strategies to find out all the CDs. We will first check the validity (with our checking algorithm introduced in Section 3.1) of the second level of the lattice which contains all the single attribute and the last level which contains a set of all the attributes as RHS. We use these nodes because they can

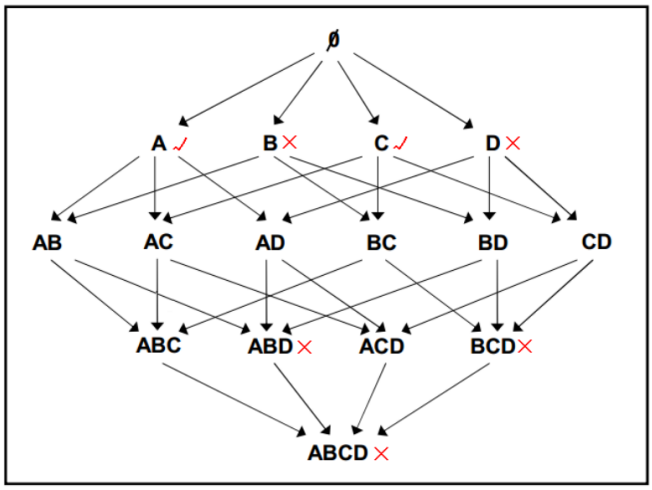

Figure 3.9: An example of lattice

provide us with some good starting points for either top-down or bottom-up mining. Also checking the validity of these nodes would not cause much overhead. Once we got the validity of these nodes, we will use them as starting points and explore the whole lattice to mine CDs.

Ideally, with these starting points and the inference rules we can validate all the nodes in the lattice. However, most of cases we cannot find out the validity of every single node in the lattice with only inference rules. So whenever our algorithm cannot proceed, we need to use the check algorithm to check an unchecked node so that the mining process could proceed.

Another problem in this situation is, when we are no longer able to find CDs with inference rules, **which** node shall we choose to verify? In our algorithm we will use a greedy strategy and always find the node that could benefit the algorithm most. That is, the nodes with most outgoing inference edges.

Data: The input table **T**, all the functional/ordering dependencies, a chosen LHS attribute R
Result: All the clustering dependencies on **T** with LHS being R
Initialization;
Construct the lattice;
Use existing ordering dependencies to discover new clustering dependencies;
For each functional/ordering dependency, add an Inference Edge into the lattice graph;
Verify the nodes on the second level and last level of the lattice;
`// S is the set that save recently validated nodes in the last step`
newValidate = {Validity of nodes in the 2nd row and the last row};
`// largestDeg is a priority queue (maximum heap) that saves the nodes and uses degree as the keys`
Generate *largestDeg* based on the degree of each nodes;
repeat
  `// First, expand with the newly validated nodes`
  while *newValidate*$\neq \emptyset$ **do**
    randomly choose a node p from newValidate;
    if *p is a valid node* **then**
      S = {valid nodes found via pruning rule 1 and 3};
      newValidate = newValidate + S;
    end
    else if *p is an invalid node* **then**
      S = {invalid nodes found via pruning rule 2 and 4};
      newValidate = newValidate + S;
    end
  end
  `// No more inference rules can help, choose a node`
  q = largestDeg.front();;
  while *q is already verified* **do**
    largestDeg.pop();
    q = largestDeg.front();
  end
  verify node p;
  mark node p as valid/invalid;
  apply according pruning rules on p;
  newValidate = newValidate + { new validated nodes via p};
until *largestDeg* $= \emptyset$;

Algorithm 2: CD mining algorithm.

Our mining algorithm takes place on a lattice composed with a set of attributes except for the selected LHS attribute. The goal is to find out the validity of every node in the lattice and record them. An example with the attribute set being $\{A, B, C, D\}$ is shown in Figure 3.9. We can see from the figure that at this stage of the algorithm we have found out the validity of 7 nodes.

To better demonstrate our algorithm, we will introduce the following concepts:

Definition 3.6. An **Inference Graph** contains all the unverified nodes in the lattice and the relations between them. It is created when the algorithm starts.

Definition 3.7. Vertices of Inference Graph Every node in the lattice is splitted into two counterparts. One indicates the node being valid (denoted with $(True)$), the other indicates the node being invalid (denoted with $(True)$). Both nodes will be placed into the inference graph as **vertices**.

Definition 3.8. A **Knowledge Pool** of the algorithm contains the following three items:

1. All the FDs passed as inputs to the algorithm.
2. All the ODs passed as inputs to the algorithm.
3. Every verified nodes in the lattice and their status.

Definition 3.9. Edges of Inference Graph: Given two nodes N_1 and N_2 in inference graph, there will be an edge pointing from N_1 to N_2 iff we can infer N_2 from N_1 and the information in knowledge pool.

The main part of the algorithm in shown in Algorithm 2. The steps are:

1. We will use our CD checking algorithm to check the second and last level of the lattice. We will then add the validity of these nodes into our knowledge pool. For example, if our $S = \{A, B, C, D\}$, we will verify A, B, C, D and $ABCD$ first.
2. We will build our inference graph based on the knowledge pool. An example of inference graph is shown in Figure 3.10. Take node BC in the lattice for example, we will split it into two vertices: BC(True) and BC(False) and put them into our inference graph.

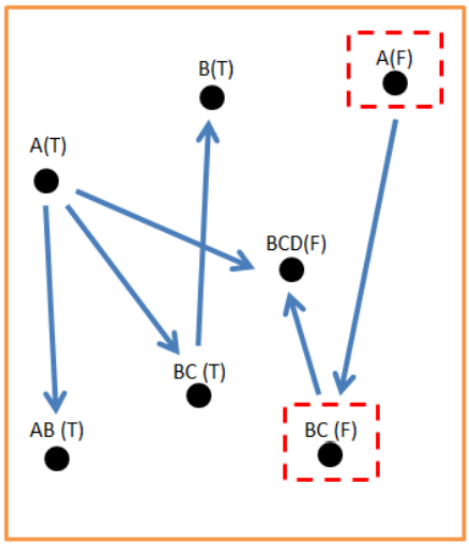

Figure 3.10: An example of an inference graph.

3. We will choose the node that has the most outgoing edges and use CD checking algorithm to verify its validity. After that, we will get the validity of this node and all the nodes it connects to (either directly or indirectly). We will add this information into our knowledge pool as well. Whenever we get the validity of a node, we will remove its counterpart from the inference graph. For example in Figure 3.10, as long as we have verified that A(True) is true. We will add all the nodes it connects to (AB(True), BC(True), B(T), BCD(False)) into our knowledge pool and will consider A(False) to be wrong and remove the node as well as the related edges from the inference graph.

4. If the inference graph is empty (which means we have verified or inferred every node in the lattice), terminate the algorithm and return the knowledge pool. Otherwise, go back to (3).

Chapter 4

Experiments

In this chapter, we conduct a series of experiments that test the performance of the algorithms we proposed. Since the complexity for checking algorithm is pretty straightforward and based on the fact we already proved its correctness, it would be unnecessary to test the CD checking algorithm here individually. Also, they can be tested within the mining algorithm.

For the CD mining algorithm, we will propose two ways to test it. First, we will generate some arbitrarily data manually according to input arguments. We uses them to test the performance of our algorithm w.r.t. different type of data. Second, we will use real-life datasets to test the performance of our CD mining algorithm, especially the pruning strength. We will first use the SourceForge Research Data Archive (SRDA), a Repository of FLOSS Research Data. We will also use a real-life table that contains order information from Superstore, as introduced in Chapter 1.

4.1 Testing on Synthetic Data

4.1.1 Preliminaries

Before we start testing our algorithm with real-life data, we first test our algorithm using synthetic data we generated. We use synthetic data first because we want to fully test the performance of our algorithm from every different aspects.

We will take the following 5 parameters into account when we generate the data: **N, M, FD_count, OD_count, CD_count**.

- **N** represents the number of rows (tuples) in the table. This number could be sufficiently large as supported by memory.

- **M** represents the number of columns (attributes) in the table. This number would usually be rather small because it could make the time complexity of the algorithm grow exponentially. Our program could support M being as large as the word length of the machine (64, most of the cases), although that could take a lot of time and thus make the testing part much harder. As a result, in our experiment we usually make M no larger than 20.

- **FD_count** indicates the number of functional dependencies (FDs) we plant into our table to help the mining process. We do this because naturally if the values in the table are randomly generated, it is very unlikely that we could discover FDs from the table, unless we deliberately plant some in. Here we only consider FDs with one LHS attribute for simplicity. These FDs are of the form: $A \rightarrow B$, where A and B could be any attribute in the table.

- **OD_count** the number of ordering dependencies (CDs) we plant into our table to help the mining process for the same reason above.. Likewise, we are only considering ODs with one attribute on both LHS and RHS for simplicity. These ODs are of the form: $\{A^{<}\} \rightsquigarrow \{B^{\leqslant}\}$ (could also be $\{A^{<}\} \rightsquigarrow \{B^{=}\}$ or $\{A^{<}\} \rightsquigarrow \{B^{<}\}$, varies when we are actually generating data) and $\{A^{=}\} \rightsquigarrow \{B^{=}\}$, where A , B can be any attribute from the table.

- **CD_count** represents the number of clustering dependencies (CDs) we are plant into the table. They are of the form $A \mapsto \{B_1, B_2, ..., B_k\}$, $k \leqslant m$, A and B_i $(1 \leqslant i \leqslant k)$ are all attributes from the table. We add this parameter to make sure there would be at least **CD_count** clustering dependencies in the table. This parameter can be used to test how number of CDs could affect the mining process. Notice that **CD_count** doesn't necessarily mean that we only have that many clustering dependencies in the table. In fact, we could always find more than that, especially when we have several FDs/ODs are planted ahead. Finally, unlike the FDs and ODs we planted, these pre-made CDs are not unknown to the mining algorithm beforehand.

4.1.2 Data Generation

The following shows how we generate the synthetic data:

First we will just fill this $N * M$ table with random numbers. The tricky part is, we don't want it to be "too random". Otherwise, as the number of

# of rows	# of cols	# of FDs	# of ODs	# of CDs	Proportion of optimized nodes	Running time	Running time (Naïve)	Improvement over Naive
10000	10	1	1	1	70.51%	2.64s	8.69s	69.62%
10000	10	2	2	2	89.97%	1.25s	9.92s	87.37%
10000	14	1	1	1	59.07%	91.56s	292.50s	68.87%
10000	14	2	2	2	85.09%	27.87s	180.79s	84.58%
10000	16	1	1	1	53.45%	382.36s	765.43s	50.08%
10000	16	2	2	2	76.33%	201.47s	803.84s	74.94%
10000	16	3	3	3	89.81%	88.5s	789.29s	88.79%

Figure 4.1: General performance test

columns grows, we can have enormous number of clustering dependencies, because every row tend be unique from each other, which makes them keys of the table. As a result, the table would be clustered on each individual set of attributes, these dependencies are considered neither interesting or useful to us. Thus when we generate data for each row, it would be identical to some previous row by a slight chance.

After that, we will generate all the FDs/ODs/CDs as required and plant them into the table. The affected attributes are randomly chosen. When we generate these dependencies, we need to coordinate between them and make sure there would be no conflicts or potential corruption.

Recall the time complexity of our checking algorithm is $O(MN)$ in the worst case for a single CD candidate. The whole search space(potential number of CDs) with a given LHS is 2^M, where N is the number of rows and M is the number of attributes. With a given LHS attribute, an naive mining algorithm that checks every potential CD could have $O(MN2^M)$ as worst-case time complexity. We will use this naive algorithm in the experiments as comparison to our mining algorithm to show the pruning strength of the algorithm.

4.1.3 General Test

First, we will generally test the performance of our algorithm to test the strength of our pruning. The result in shown in Figure 4.1.

As we can see here, the performance of our algorithm can and will be greatly improved when given some existing dependencies. The more existing dependencies we have, the more time we could save. Generally, we can improve the performance by $\sim$100% to $\sim$1000%.

# of rows	# of cols	# of FDs	# of ODs	# of CDs	Proportion of optimized nodes	Running time	Running time (Naïve)	Improvement over Naive
100	12	1	1	1	49.04%	0.25s	0.48s	48.30%
1000	12	1	1	1	70.05%	1.49s	4.93s	68.96%
2000	12	1	1	1	66.03%	2.85s	8.18s	65.03%
5000	12	1	1	1	59.02%	8.55s	21.17s	59.68%
10000	12	1	1	1	63.28%	15.58	42.84s	63.49%
20000	12	1	1	1	62.34%	32.17s	84.95s	62.10%

Figure 4.2: Scalability test w.r.t. number of rows.

# of rows	# of cols	# of FDs	# of ODs	# of CDs	Proportion of optimized nodes	Running time	Running time (Naïve)	Improvement over Naive
3000	10	1	1	1	73.31%	0.71s	2.60s	72.58%
3000	12	1	1	1	66.18%	4.05s	11.79s	65.66%
3000	14	1	1	1	66.17%	18.04s	51.54s	65.02%
3000	16	1	1	1	60.35%	89.32s	225.09s	60.22%
3000	18	1	1	1	50.48%	472.04s	874.41s	51.56%

Figure 4.3: Scalability test w.r.t number of columns.

4.1.4 Scalability Test

Now we will test the scalability of our algorithm, first w.r.t. the number of rows **N**. The result in shown in Figure 4.2.

As we can see here, the time cost grows linearly according to the number of rows. We can see that our algorithm exhibits good scalability on number of rows.

Second, we will test the scalability of our algorithm w.r.t. the number of columns. The result in shown in Figure 4.3.

As we can see here, the time grows exponentially as the number of columns grows. Specifically, the time cost doubles when the number of columns is incremented by 1 as expected. As the time complexity is strongly related to the size of the lattice which grows exponentially w.r.t. the number of attributes.

4.1.5 Test of Effectiveness of Different Dependencies

We also test how planting different types of dependencies could affect the performance of the algorithm. We experiment with varies combination of the number of each dependency, with the results shown in Figure 4.4.

The first thing we find out in the results is that even though adding

# of rows	# of cols	# of FDs	# of ODs	# of CDs	Proportion of optimized nodes	Running time	Running time (Naïve)	Improvement over Naive
3000	12	2	0	1	49.02%	6.21s	11.76s	47.27%
3000	12	0	2	1	78.14%	2.65s	11.81s	77.55%
3000	12	0	0	3	0.2604%	12.16s	12.48s	N/A
3000	16	3	0	2	51.81%	50.58s	101.25s	50.11%
3000	16	0	3	2	91.48%	8.96s	105.99s	91.53%
3000	16	0	0	5	0.2380%	107.30s	106.11s	N/A

Figure 4.4: Test on effectiveness of different dependencies.

functional dependencies could already benefit the performance of the algorithm, adding ordering dependencies acts even better. This is reasonable since ordering dependencies could imply functional dependencies. Moreover, they are specialized so that the LHS are exactly the LHS of the clustering dependencies we want to discover. Another thing to see is that planting in nothing but clustering dependencies almost does no help to our algorithm. This is also acceptable because first unlike functional/ordering dependencies, pre-defined clustering dependencies are not known to our program beforehand. Perhaps another reason is that the union rules themselves are not quite effective in pruning.

4.2 Testing on Real Data

In this section, we will experiment with two sets of real-life data. First, we will test with the SourceForge Research Data Archive (SRDA) data. The data made available from this FLOSS research data archive, is derived from and NSF funded project, entitled "Understanding Open Source Software Development". This research project seeks to understand the free/open source software (F/OSS) phenomenon and to predict the pattern of growth exhibited by F/OSS projects over time. [15][16][17]

In the pre-processing step, we will first use FD/OD mining algorithms to discover all such dependencies and pass them to our CD mining algorithm. We already have fairly efficient algorithms for function dependency mining like [12]. But there is no valid algorithm to discover ordering dependencies. However, consider this is just pre-processing and our data size is not very large, we are able to endure the time cost and implement a naive algorithm to find out all the ODs.

We specifically look at the following 6 tables: **doc_data**, **forum**, **people_job_inventory**, **artifact_group_list**, **artifact**, **artifact_file**.

FileName	#Rows	#Lines	#FDs	#ODs	MiningTime(s)	MiningTimeNaive (s)	OptimizedNodes	TimeSaved
doc_data	16985	7	17	6	24.78	33.53	62.50%	26.10%
forum	47912	8	7	0	186.08	246.96	57.03%	24.65%
people_job_inventory	21725	5	4	0	28.08	43.84	25%	35.95%
artifact_group_list	17326	11	29	12	608.85	831.71	73.83%	27.80%
artifact	33787	12	12	0	3825	4791.84	14.75%	20.18%
artifact_file	21340	7	6	0	31.85	40.11	46.88%	20.59%

Figure 4.5: Test on SRDA dataset

The results are shown in Figure 4.5.

Within these datasets, there are not a lot of ODs we can make use of, and most of the FDs are true because the first column in the primary key. Those FDs are generally not helpful to us.

As we can see, most of the time, our mining algorithm can prune the search space just fine, especially with the help of ordering dependencies. However the performance overall in not quite satisfying. The reason is that there are not many clustering dependencies in these tables and the pruning strategy was used less than expected. Regardless, our mining algorithm can still improve the performance by at least 20% even without the help of any additional FDs or ODs.

As a second set of real-life data, we perform tests using the orders information table from Superstore introduced in Chapter 1.

Recall that the table has 21 attributes: **Row ID, Order ID, Order Date, Ship Date, Ship Mode, Customer ID, Customer Name, Segment, Country, City, State, Postal Code, Region, Product ID, Category, Sub-Category, Product Name, Sales, Quantity, Discount and Profit**. Plus it has 9993 records. A table with such size could make the time complexity of mining algorithm unbearable. Thus we need to conduct some pre-processing to reduce the search space.

First, if we study the semantic of these attributes, we will be able rule out some attributes that has no potential involvement with clustering dependencies. Namely, **Product Name, Sales, Quantity, Discount, Profit and Country (Since nearly all of them are from United States)**. That provide us with 15 attributes left.

Second, instead of testing on 15 columns directly, we will test on the first 8 columns, then on 12 columns and finally on 15 columns. Besides, we also distinguish the case where LHS being **(Row) ID** from others. We do this because most FDs are involved with that attribute. Thus we can show how we can greatly reduce the search space with the help of existing FDs. For

# of columns	LHS column	# of FDs	# of ODs	# of CDs (TOTAL)	MiningTime(Naive) (seconds) (AVG)	MiningTime (seconds) (AVG)	TimeSaved
8	**ID**	13	0	35	3.40	1.01	70.294%
8	*others*	13	0	5	2.71	2.02	25.461%
12	**ID**	24	0	764	62.62	18.29	70.792%
12	*others*	24	0	255	44.45	37.32	16.040%
15	**ID**	31	0	1540	409.02	28.94	92.925%
15	*others*	31	0	521	347.75	252.76	27.316%

Figure 4.6: Test on Order table

the LHS being other attributes, we just show the result of the average cost since their performance is quite similar. The results are shown in Figure 4.6.

As we can see here, since most of the FDs are involved with the attribute **(Row) ID**. Based on the fact that there are a great number of CDs with **(Row) ID** on the LHS, the search space would be greatly pruned. When we include the all 15 attributes, we can save as much as 92.925% of the time compared to the naive algorithm. We can assure that by definition, all of these CDs are interesting. On the other hand, clustering dependencies with other LHS do not have as good performance. For example, with other 14 attributes being the LHS, the number of valid CDs sums up to only 521. Accordingly, pruning for them would be much weaker.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we first introduced clustering dependencies (CDs) and demonstrated with examples how it is useful in different fields including query optimization, data visualization, data analysis, MapReduce, etc. After that, we gave the formal definition to clustering dependencies. We also discussed relation of CDs to functional dependencies (FDs) and ordering dependencies (ODs).

We then discussed reasoning for clustering dependencies. We declared that although it is difficult to come up with a sound and complete system, we will still be able to discover a lot of inference rules that turned out to be very useful in the CD mining algorithm. We then exhibited some inference rules with the help of functional dependencies (FDs) and ordering dependencies (ODs).

After that, we introduced two algorithms, a checking algorithm that can check the validity of a given CD. It has a $O(MN)$ time complexity in the worst case. This algorithm runs by checking RHS attributes of every tuple in the given CD and returns when a violation occurs. We also defined uninteresting clustering dependencies to rule out the CDs that, for example, have keys on the RHS. The second algorithm we proposed is a mining algorithm that can discover all the CDs within a given table. The mining algorithm works with the help of inference rules and the checking algorithm. We showed that we can make good use of the inference rules to prune the search space.

Finally, we tested the performance of our mining algorithm with both synthetic and real-life datasets. With synthetic data, we observed that the

time cost of the algorithm followed our experimental analysis. We have also experimentally demonstrated that our algorithm is highly scalable. Cohering the real-life datasets, we showed that not only can we greatly prune the search space, especially with the help of FDs and ODs, but we can also discover a lot of interesting CDs from real-life datasets.

5.2 Future Work

We propose the following as overviews of potential future work:

1. We will consider more expressive and comprehensive LHS form instead of just one single attribute. We have actually considered using multiple attributes on the LHS, but this complicates the development of CD. We also believe that having a set of LHS attributes **ORDER BY** is much better than having the LHS attributes **GROUP BY**, simply because the former would be subsumed by the latter. However, using multiple attributes on the LHS would make the LHS become an OD, in which case we have to make extra effort to deal with OD's reasoning system and mining algorithm. Unfortunately, although the reasoning system for ODs have already been established quite well lately, there haven't been any well-known efficient OD mining algorithm yet. Even if there is, the question of whether ODs with a lot of attributes on both sides would be interesting or worth studying at all could be highly questionable.

2. We will try to relax the logics in our definition in the hope that it could bring us more interesting types of CDs. We will put CDs in this category as **Almost GROUP BY**. There are two reasons for this. First, the data might contain noise (wrong records), and those few tuples could ruin the whole CD integrity. Secondly, when the size of the data is really large, we might need to do some filtering upon it, in which case we only need the data to be approximately aggregated.

3. We will try to further discover more inference rules and try to build up a more comprehensive and sophisticated reasoning system, although it would be much harder than it appears to be. The ultimate goal would be a discovery of a complete set of inference rules for CD reasoning.

4. We will try to improve the performance of our mining algorithm. We believe the key is to prunes the CD candidates with very large RHS.

It could be the case that when the size of RHS attributes, or the singularity of the RHS attribute value combinations reaches a certain threshold, we will no longer be interested in them. Also if the RHS contains some specific features, we will consider any further RHS containing those attributes uninteresting directly. Besides, we could also consider developing different strategies for different types of input data.

Bibliography

[1] Bitton, Dina, Jeffrey Millman, and Solveig Torgersen. “A feasibility and performance study of dependency inference [database design].” Data Engineering, 1989. Proceedings. Fifth International Conference on. IEEE, 1989.

[2] Mannila, Heikki, and Kari-Jouko RÃďihÃď. “Algorithms for inferring functional dependencies from relations.” Data Knowledge Engineering 12.1 (1994): 83-99.

[3] Huhtala, YkÃď, et al. “TANE: An efficient algorithm for discovering functional and approximate dependencies.” The computer journal 42.2 (1999): 100-111.

[4] Szlichta, Jaroslaw, Parke Godfrey, and Jarek Gryz. “Fundamentals of ordering dependencies." Proceedings of the VLDB Endowment 5.11 (2012): 1220-1231.

[5] Szlichta, Jaroslaw, et al. “Expressiveness and complexity of ordering dependencies.” Proceedings of the VLDB Endowment 6.14 (2013): 1858-1869.

[6] Wyss, Catharine, Chris Giannella, and Edward Robertson. “Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract.” Data Warehousing and Knowledge Discovery. Springer Berlin Heidelberg, 2001. 101-110.

[7] Huhtala, YkÃď, et al. “TANE: An efficient algorithm for discovering functional and approximate dependencies.” The computer journal 42.2 (1999): 100-111.

[8] Dong, Jirun, and Richard Hull. “Applying approximate ordering dependency to reduce indexing space." Proceedings of the 1982 ACM SIGMOD international conference on Management of data. ACM, 1982.

[9] Ginsburg, Seymour, and Richard Hull. “Order dependency in the relational model." Theoretical computer science 26.1 (1983): 149-195.

[10] Armstrong, William Ward. “Dependency Structures of Data Base Relationships." IFIP congress. Vol. 74. 1974.

[11] Fan, Wenfei, et al. “Discovering conditional functional dependencies." Knowledge and Data Engineering, IEEE Transactions on 23.5 (2011): 683-698.

[12] Chiang, Fei, and RenÃľe J. Miller. “Discovering data quality rules." Proceedings of the VLDB Endowment 1.1 (2008): 1166-1177.

[13] Golab, Lukasz, et al. “On generating near-optimal tableaux for conditional functional dependencies." Proceedings of the VLDB Endowment 1.1 (2008): 376-390.

[14] Chu, Xu, Ihab F. Ilyas, and Paolo Papotti. “Discovering Denial Constraints." Proceedings of the VLDB Endowment 6.13 (2013).

[15] Matthew Van Antwerp and Greg Madey, “Advances in the SourceForge Research Data Archive (SRDA)", The 4th International Conference on Open Source Systems, IFIP 2.13 - (WoPDaSD 2008), Milan, Italy, September 2008. (paper) (slides) (BibTeX citation)

[16] Yongqin Gao, Matthew Van Antwerp, Scott Christley and Greg Madey, “A Research Collaboratory for Open Source Software Research", In the Proceedings of the 29th International Conference on Software Enginering + Workshops (ICSE-ICSE Workshops 2007), International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS 2007), Minneapolis, MN, May 2007. (paper)

[17] Greg Madey, ed., The SourceForge Research Data Archive (SRDA). University of Notre Dame. May 2014 http://srda.cse.nd.edu/

[18] Business Intelligence and Analytics | Tableau Software, http://www.tableau.com/

[19] e-Study Guide for: Integrated Business Projects by Anthony A. Olinzock, ISBN 9780538731096

[20] Steele, Julie, and Noah Iliinsky. “Beautiful visualization: looking at data through the eyes of experts." O’Reilly Media, Inc., 2010.

[21] Dean, Jeffrey, and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.

[22] “Google spotlights data center inner workings", Tech news blog - CNET News.com.

[23] Big Data Infrastructure, CS 489/698 (Winter 2016), University of Waterloo. http://lintool.github.io/bigdata-2016w/.

[24] Baudinet, Marianne, Jan Chomicki, and Pierre Wolper. “Constraint-generating dependencies." Database Theory - ICDT’95. Springer Berlin Heidelberg, 1995. 322-337.